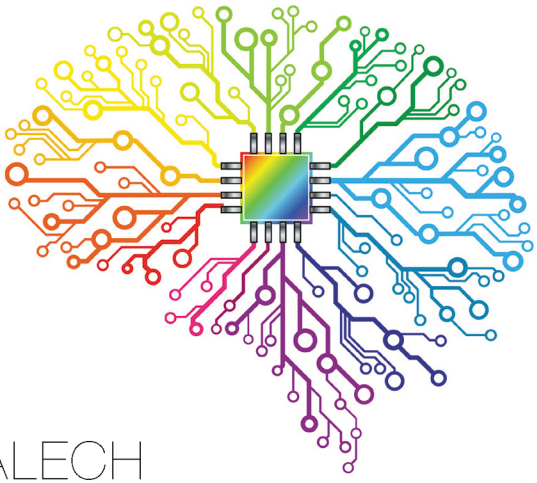
ARTIFICIAL INTELLIGENCE METHODS FOR SOFTWARE ENGINEERING



EDITORS

MEIR KALECH

RUI ABREU

MARK LAST

World Scientific

ARTIFICIAL INTELLIGENCE
METHODS FOR SOFTWARE ENGINEERING

This page intentionally left blank

ARTIFICIAL INTELLIGENCE METHODS FOR SOFTWARE ENGINEERING

Editors

Meir Kalech
Ben-Gurion University of the Negev, Israel

Rui Abreu
University of Porto, Portugal

Mark Last
Ben-Gurion University of the Negev, Israel

World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI • TOKYO

Published by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

This book has been partially funded by the Cyber Security Research Center at Ben-Gurion University of the Negev and by ISF grant No. 1716/17. This material is based upon work supported by Fundação para a Ciência e a Tecnologia (FCT), with the reference PTDC/CCI-COM/29300/2017 and UID/CEC/50021/2019. The authors further would like to thank Roni Stern for being a research partner.

ARTIFICIAL INTELLIGENCE METHODS FOR SOFTWARE ENGINEERING

ISBN 978-981-123-991-5 (hardcover)
ISBN 978-981-123-992-2 (ebook for institutions)
ISBN 978-981-123-993-9 (ebook for individuals)

For any available supplementary material, please visit
https://www.worldscientific.com/worldscibooks/10.1142/12360#t=suppl

Printed in Singapore

To my lovely wife Ravit,
our children - Oran, David, Barak and Zohar, and
my parents Mira and Eliko.
M.K.

To my lovely wife Liliana, and our children
Duarte, Bernardo and Filipe.
R.A.

To my beloved wife Tami, my parents Rosa
and Isidore, and our children Dan, Iris, and Einat.
M.L.

This page intentionally left blank

https://doi.org/10.1142/9789811239922_fmatter

Preface

Back in 2011, Marc Andreessen, co-founder and general partner of venture capital firm Andreessen Horowitz, wrote an essay[1] in The Wall Street Journal on the fact that "*Software is eating the world.*" A couple of years later, in 2014, Dutch computer scientist and entrepreneur Erik Meijer, since 2015 a Director of Engineering at Facebook, co-authored a paper published in the Communications of the ACM[2] corroborating the same thought: "of the *top five fastest-growing companies with regard to market capitalization in 2014, three are software companies: Apple, Google, and Microsoft (in fact, one could argue that Intel is also driven by software, making it four out of five).*" Arguably, over the last couple of decades, software technology has been one of the primary drivers of economic growth in the world. Developing reliable software, however, remains far from trivial. A 2013 study[3] from Cambridge University estimates that software bugs cost the global software Industry a staggering $316 billion per year. As software becomes one of the fundamental pillars of almost any company, following engineering concepts for software designing, creating, improving, and maintaining software becomes of paramount importance.

Software Engineering as a discipline is still evolving (the field is a merely shy of 50 years old[4]), with the emergence of agile practices, the need to consider technology management, legacy software integration, organizational management, as well as deployment and infrastructure issues instead of a focus on just developing software code. Successful software projects,

[1] https://a16z.com/2011/08/20/why-software-is-eating-the-world/

[2] Erik Meijer and Vikram Kapoor. "The responsive enterprise: embracing the hacker way." Communications of the ACM 57.12 (2014): 38–43.

[3] Research by Cambridge MBAs for tech firm Undo finds software bugs cost the industry $316 billion a year: https://goo.gl/mikn7P

[4] The NATO Software Engineering Conferences were held in 1968 and 1969, being attended by international experts on computers who agreed on defining best practices for software grounded in the application of engineering. The conferences played a major role in gaining general acceptance for the term software engineering.

therefore, require more than just technical expertise: understanding the real needs of different stakeholders, collaborating in a team (nowadays, potentially globally distributed), managing complexity, mitigating risks, delivering projects on time and on budget, and determining when a software product is good enough to be shipped are at least equally important topics that often have a significant human component (the so-called soft skills).

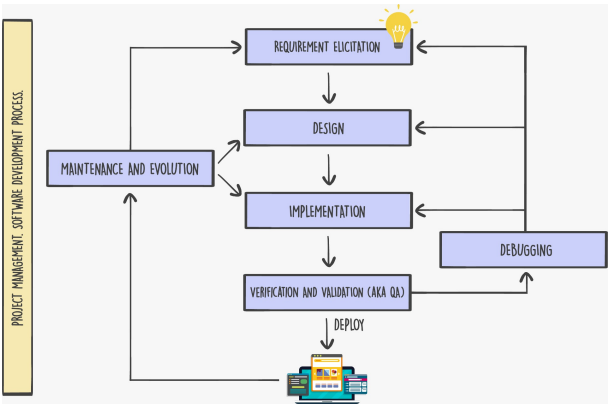
As Alexander Pope stated: "to err is human" (following the famous Latin proverb "errare humanum est" attributed to Hieronymus). Therefore, software as a human-made component is naturally prone to errors and/or inconsistencies. Ascertaining that no human errors creep into the different phases of the software lifecycle is where artificial intelligence can be a game-changer. It is already a fact that Artificial Intelligence can simplify several tasks, ranging from interacting with our consumer electronic appliances to playing our favourite music. Bringing together Artificial Intelligence and Software Engineering has the potential to create high quality software artifacts, hence reducing time-to-market, while maintaining high standards.

The history of the field of Artificial Intelligence dates back to the seminal work of Turing and McCarthy. The gist of Artificial Intelligence is about making machines intelligent in order for them to perform tasks that are rather complex for humans. Software engineering is, in fact, one of the most challenging of all engineering disciplines (despite it is often not recognised as such). Therefore, researchers and industrialists alike have sought to apply Artificial Intelligence methods to solve Software Engineering problems. In fact, back in 1986, Simon argued that "*it is not really a question of whether we want to use artificial intelligence methods in software engineering: it is a question of whether artificial intelligence is powerful enough to help us.*"[5]

Currently, there are several developments in Artificial Intelligence methods and approaches that make them well suited to address software engineering problems. The authors of the chapters of this book discuss the advances in state-of-the-art of several software engineering problems by leveraging Artificial Intelligence methods to techniques such as Natural Language Processing (NLP), machine learning, fuzzy logic, multi-objective search, and metaheuristics.

In particular, the chapters of this book cover several aspects of the software engineering lifecycle (see image below), organized as follows:

[5]Herbert A. Simon. "Whether software engineering needs to be artificially intelligent." *IEEE Transactions on Software Engineering* 7 (1986): 726–732.



AI for Software Design

Interweaving AI and Behavioral Programming Towards Better Programming Environments by Achiya Elyasaf, Moshe Weinstock and Gera Weiss;

AI Techniques for Software Requirements Prioritization by Alexander Felfernig

Agent-Based Software Programming

Social Commitments for Engineering Interaction in Distributed Systems by Matteo Baldoni, Cristina Baroglio, Roberto Micalizio and Stefano Tedeschi

Intelligent Agents are More Complex: Initial Empirical Findings by Gal A. Kaminka and Alon T. Zanbar

AI for Software Development

Sequence-to-Sequence Learning for Automated Software Artifact Generation by Zhongxin Liu, Xin Xia and David Lo

Machine Learning to Support Code Reviews in Continuous Integration by Miroslaw Staron, Mirosław Ochodek, Wilhelm Meding, Ola Söder and Emil Rosenberg

Software Fusion: Deep Design Learning With Deterministic Laplacian Verification by Iaakov Exman

Using Artificial Intelligence for Auto-Generating Software for Cyber-Physical Applications by Gregory Provan

AI for Software Testing

On the Application of Machine Learning in Software Testing by Nour Chetouane and Franz Wotawa

Creating Test Oracles Using Machine Learning Techniques by Rafig Almaghairbe and Marc Roper

Intelligent Risk Based Analysis Methodology by Eli Menasheof

A Qualitative Reasoning Model for Software Testing, based on Combinatorial Geometry by Spyros Xanthakis and Emeric Gioan

AI for Software Debugging

AI-based Spreadsheet Debugging by Konstantin Schekotihin, Birgit Hofer, Franz Wotawa and Dietmar Jannach

Artificial Intelligence Methods for Software Debugging by Wolfgang Mayer and Franz Wotawa

This book focuses on the above topics since these are the building blocks of every software. The AI techniques presented by the papers in this book could leverage the design process by making it faster and more organized, the development process more comfortable, and the testing and debugging processes more reliable.

All chapters published in this book have undergone editorial review as well as external anonymous peer review. We would like to thank all contributors and reviewers for their diligent work in bringing together this exciting book. We would also like to thank those who have helped us putting this book together.

Meir Kalech, Associate Professor
Department of Software and Information Systems Engineering
Faculty of Engineering Sciences
Ben-Gurion University of the Negev
Israel
kalech@bgu.ac.il

Rui Abreu, Full Professor
Department of Informatics Engineering
Faculty of Engineering
University of Porto
Portugal
rui@computer.org

Mark Last, Full Professor
Department of Software and Information Systems Engineering
Faculty of Engineering Sciences
Ben-Gurion University of the Negev
Israel
mlast@bgu.ac.il

This page intentionally left blank

https://doi.org/10.1142/9789811239922_fmatter

Contents

Preface vii

AI for Software Design **1**

1. Interweaving AI and Behavioral Programming Towards Better Programming Environments 3
   Achiya Elyasaf, Moshe Weinstock and Gera Weiss

2. AI Techniques for Software Requirements Prioritization 29
   Alexander Felfernig

Agent-Based Software Programming **49**

3. Social Commitments for Engineering Interaction in Distributed Systems 51
   Matteo Baldoni, Cristina Baroglio, Roberto Micalizio and Stefano Tedeschi

4. Intelligent Agents are More Complex: Initial Empirical Findings 87
   Gal A. Kaminka and Alon T. Zanbar

AI for Software Development **109**

5. Sequence-to-Sequence Learning for Automated Software Artifact Generation 111
   Zhongxin Liu, Xin Xia and David Lo

6. Machine Learning to Support Code Reviews in Continuous Integration 141
Miroslaw Staron, Miroslaw Ochodek, Wilhelm Meding, Ola Söder and Emil Rosenberg

7. Software Fusion: Deep Design Learning with Deterministic Laplacian Verification 169
Iaakov Exman

8. Using Artificial Intelligence for Auto-Generating Software for Cyber-Physical Applications 211
Gregory Provan

AI for Software Testing **241**

9. On the Application of Machine Learning in Software Testing 243
Nour Chetouane and Franz Wotawa

10. Creating Test Oracles Using Machine Learning Techniques 269
Rafig Almaghairbe and Marc Roper

11. Intelligent Risk Based Analysis Methodology 305
Eli Menasheof

12. A Qualitative Reasoning Model for Software Testing, based on Combinatorial Geometry 331
Spyros Xanthakis and Emeric Gioan

AI for Software Debugging **369**

13. AI-based Spreadsheet Debugging 371
Konstantin Schekotihin, Birgit Hofer, Franz Wotawa and Dietmar Jannach

14. Artificial Intelligence Methods for Software Debugging 401
Wolfgang Mayer and Franz Wotawa

Index 437

This page intentionally left blank

AI for Software Design

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0001

Chapter 1

Interweaving AI and Behavioral Programming Towards Better Programming Environments

Achiya Elyasaf*, Moshe Weinstock† and Gera Weiss‡
Ben-Gurion University of the Negev, Israel
*achiya@bgu.ac.il, †moshewe@cs.bgu.ac.il, ‡geraw@cs.bgu.ac.il

1.1 Introduction

A key challenge in developing reactive systems like robots and smart machines is coping with the huge variety of scenarios that may arise when the machine or the robot runs in a rich non-deterministic environment. Scenario-based specification and programming techniques cope with this challenge by providing engineers with the tools to specify independent patterns and anti-patterns of behavior. These patterns are interwoven at runtime to match the actual scenario that the robot or the machine has to deal with. This approach has been proven to be useful in a variety of application domains. A key feature of behavioral-based specifications is that the resulting specification is often non-deterministic in the sense that it gives the robot a "freedom of choice" in many situations, especially at early development stages, when the library of patterns is not complete. Since it is desirable to be able to explore the executions of the models, even in these early stages, researchers and industry have proposed tools to resolve this nondeterminism by means of synthesis, model-checking, planning, machine learning, and priorities. The challenge here is to propose ways to make informed decisions in limited time and to reuse data from earlier situations when applicable.

In this chapter we present our ongoing work towards resolving the nondeterminism in scenario-based specifications by running reinforcement

learning and online evolutionary algorithms that traverse possible continuations of execution. We describe mechanisms that, when the robot has a choice of action, consider possible future actions of both the environment and the robot, as well as decide on a "best" action for the robot. With these mechanisms, programmers and other stakeholders can provide robots with programs (scenario-based specifications) that specify most of the behavior while leaving enough choice for the program to cope with the dynamicity in the environment.

To demonstrate our approach, we outline our work on three case studies, where in all cases, we developed a decisions making controller for a simulated robotic software. Of course, while the examples demonstrate robotic applications, our approach is general, intended to be used for any reactive system. Moreover, the behavioral programming paradigm that we base our approach on, has been used for modeling and programming other types of reactive systems, such as smart-home automation [1], a satellite mission controller [2], biological processes [3] and many more.

Unlike pure AI approaches, our goal is to support software engineers, not to replace them. Specifically, in a software engineering context, where it is mandatory that programmers have full control of the behavior of the system, AI can only be used when its actions follow the commands of the programmers. Our goal is to ease the job of the programmers by relieving the pain of taking care of all the gory details. As an illustrative, non-technical example, consider the difference between a car driver and a coachwoman. The car driver must provide her car with precise steering commands while the coachwoman has at her disposal an intelligent agent (the horse) that can implement partial or incomplete steering commands. Rephrasing this in a technical setting, we demonstrate in this chapter that programmers of complex reactive systems can benefit from having an intelligent program execution mechanism that can execute partial behavioral specifications. Specifically, programmers can specify behavioral patterns and anti-patterns for enforcing critical safety requirements. This way, the execution mechanism has enough room to take care of non-functional requirements, based on cost functions and on a model of the environment. Note that, as software engineers, we are not aiming at taking the control out of the hands of the programmer. On the contrary, we envision an incremental development, where each iteration consists of running the model with the smart execution mechanism and of refining the program so that it does what the programmer wants it to do, a process that resembles, in some sense, the process of training a horse.

1.2 Preliminaries

Our approach is based on a model-driven engineering paradigm, called behavioral programming. We begin with a general description of these terms and their related work, and elaborate on the BP paradigm in Sec. 1.3.

1.2.1 *Behavioral and Scenario-Based Programming*

A widely accepted practice in software development is to formalize requirements in the form of use-cases and scenarios [4]. The programming approach termed *behavioral programming* (BP) extends this approach to using scenarios for the actual coding of software (executable specifications). Specifically, the method introduces scenario coding techniques and design approaches for constructing reactive systems [5] incrementally from their desired and undesired behaviors. The work on behavioral programming began with scenario-based programming, a way to create executable specifications of reactive systems, introduced through the language of *live sequence charts* (LSC) and its *Play-Engine implementation* [6,7]. The initial purpose was to enable testing and refining specifications and prototypes, and the approach was later extended towards building actual systems [8]. To this end, the underlying behavioral principles have also been implemented in imperative programming languages, via, e.g., the BPJ package [8] adding a more conventional programming point of view to that of requirement specification. Following this direction there are several tools supporting the behavioral programming principles in other languages, such as Erlang [9], C [10], and with graphical tools such as the Play-Engine [7], PlayGo [11], and SBT [12]. Our approach to scenario-based specification is based on the LSC language [6, 7]. It allows GUI-based or natural language-based playing in of behavioral scenarios, and is multi-modal, allowing constraints (for example, forbidden scenarios) to be part of the program. The approach has been generalized and extended also to other languages, including Java, C++, Erlang, JavaScript, and Blockly, and was termed behavioral programming (BP) [13]. Research results cover, among others, run-time lookahead (smart playout) [14], model-checking [15], compositional verification [16], synthesis [12], interactive analysis of unrealizable specification [17], abstraction-refinement mechanism [17], automatic correction tools [18], and synchronization relaxation tools [19]. In this chapter we present our ongoing work, on extending the approach proposed in [14] towards using self-reflection and a model of the environment in an execution mechanism that allows for more robust and AI-based adaptive behavior.

1.2.2 *Model-Driven Engineering*

Model-driven engineering and software development [20, 21] is geared to allow human stakeholders to develop and analyze a system using abstractions and formalisms that are more closely aligned with their mental models than standard programming language code. MDE is supported by many languages, platforms, and tools, including, among others, UML [22] and SysML [23], AADL[Feiler2006ArchitectureAnalysis], MATLAB Simulink [24], SCADE [25], Ptolemy II [26], and Eclipse Modeling Framework [27]. The models, built in modeling languages, including domain-specific modeling languages and platforms, can be transformed into running code through automated model transformations. In this chapter we take these ideas further, along the lines outlined in, e.g., [28], to using models also for online search and for self-reflection.

1.2.3 *Search-Based Software Engineering (SBSE)*

Many activities in software engineering can be stated as optimization problems. Search-based software engineering (SBSE) applies metaheuristic search techniques, such as evolutionary algorithms, to software engineering problems. For instance, the following is an illustrative list of SE questions [28]:

(1) What is the smallest set of test cases that covers all branches in this program?
(2) What is the best way to structure the architecture of this system to enhance its maintainability?
(3) What is the set of requirements that balances software development cost and customer satisfaction?
(4) What is the best allocation of resources to this software development project?
(5) What is the best sequence of refactoring steps to apply to this system?

Though they may appear very different, they are all essentially *optimization* problems. As such, they are typical of the kinds of problem for which SBSE is well adapted and with which each has been successfully formulated as a search-based optimization problem. Foremost among the techniques used by SBSE practitioners is *evolutionary computation*, or *evolutionary algorithms*.

1.2.4 *Evolutionary Computation*

In the field of evolutionary computation (EC) ideas from evolutionary biology — random variation and selection — are harnessed in algorithms that are applied to complex computational problems. The origins of EC can be traced back to the 1950s and 1960s but the field has come into its own over the past two decades, proving successful in solving numerous problems from highly diverse domains [29]. EC techniques are being increasingly applied to difficult real-world problems, often yielding results that are not merely academically interesting but also competitive with the work done by creative and inventive humans [30]. Evolutionary algorithms are an excellent choice when the search space involved is huge and one wishes to search it efficiently. Indeed, as mentioned above, within the field of SBSE (and others), EC is perhaps the most-used technique. Moreover, states reoccur with small changes during a BP run. Heuristics that work well for a certain phase of the search may need some modifications over time. The fundamental EC behavior of individual improvement over generations may deal with this problem seamlessly. With these considerations in mind, EC was a natural choice for the current proposal.

An evolutionary algorithm is an iterative procedure that involves a population of individuals, each one represented by (perhaps) a finite string of symbols, known as the genome, encoding a possible solution in a given problem space. This space, referred to as the search space, comprises all possible solutions to the problem at hand. The algorithm sets out with an initial population of individuals that is generated at random or heuristically. Every evolutionary step, known as a generation, the individuals in the current population are decoded and evaluated according to some predefined quality criterion, referred to as the fitness, or fitness function. To form a new population, which will constitute the next generation, individuals are selected according to their fitness, and then transformed via genetically inspired operators, of which the most well-known are crossover and mutation. Simple crossover involves "mixing" two or more genomes to form novel offspring, and simple mutation randomly flips bits in the genomes. Continually iterating this procedure, and owing to the principle of selection of the fittest, the evolutionary algorithm may eventually find an acceptable solution, i.e., one with high fitness.

1.3 The Proposed Approach

Our approach is based on the Behavioral Programming (BP) paradigm and on Artificial Intelligence (AI). The key new ingredient is an extension of the BP execution mechanism with on-line heuristic search in program state space that allows programmers to develop nondeterministic programs while relying on a "smart" event selection mechanism to resolve nondeterminism in a heuristic manner.

Our goal is to make SBSE accessible to modelers and programmers of reactive systems, such as robotic applications and interactive applications, as idioms that integrate with standard constructs in common modeling and programming languages. This allows for natural, powerful derivation from modeling languages (such as BP) to executable systems. Specifically, we present tools and techniques that facilitate the following software development methodology:

(1) Model aspects of the required behavior using non-determinism to specify free choices in execution.
(2) Run the system using an engine that resolves non-determinism heuristically or synthesizes code.
(3) If unsatisfied with the execution's choices, extend the model by formalizing more-refined requirements.
(4) Repeat steps 2 and 3 until the behavior is satisfactory.

In addition to the refinement idioms that already exist in BP, which allow programmers to incrementally shape their software by adding modules that can both widen and narrow the set of possible behaviors of the system, our approach allows BP-based models to also contain specifications of fitness criteria for the heuristic search function that can also be refined along the above development process. Our vision is that these will be used by programmers for specifying nonfunctional requirements such as the need to preserve energy and the need to accomplish a mission as fast as possible. The execution mechanism will then try to optimize these measures within the constraints posed by the base BP code that encodes functional, safety, requirements such as "Never open the box when the robot is moving" or "Always open the flaps when landing". The specification of these fitness criteria is done inside the scenarios that constitute the BP model, giving programmers a unified language with which they can specify both hard and soft rules that the application agnostic execution mechanism can weld together into a reactive program.

The idea of "smart" execution of scenario-based specifications started in [14] and in [31] with proposals to apply, respectively, model-checking and planning algorithms for running a single super-step (the part of the run that spans between two consecutive external events) in LSC. We apply similar mechanisms in the context of a behavioral programming library embedded in an imperative programming language. Beyond running in a different setting, the main addition of our approach, when compared to these earlier contributions, is that it aims at running the "smart" event selection mechanism at run-time, on real program code rather than on a model or specification. Further more, the papers cited above dealt only with synthesizing one super step at a time, i.e., they focused on planning the next move of the system in a way that satisfies all the internal constraints. Here we widen the horizon by synthesizing strategies that consider also the possible reactions of the environment, the reactions of the system to these reactions and so forth.

As said in [32], any intelligent system that operates in a complex unpredictable environment must be reactive — that is, it must respond dynamically to changes in its environment. For simple tasks in carefully engineered domains, non-reactive behavior is acceptable; for more intelligent agents in unconstrained domains, it is not. This chapter presents an architecture for intelligent reactive systems. The advantages of the proposed environments are in terms of modularity, awareness, and robustness defined in [32] as follows:

Modularity: Systems are built incrementally from simple components that are easy to implement and to understand.

Awareness: Systems are always aware of what is happening; they are always be able to react to unexpected sensory data.

Robustness: Systems are built to behave plausibly in novel situations and when some of their sensors are inoperable or impaired.

We believe that the best way to achieve these design goals is that programmers develop models that describe the choices for the robot and use online and offline search techniques to make informed choices within these models. To realize this goal, our approach integrates BP techniques with two AI based techniques — evolutionary computation and reinforcement learning.

1.4 Sandboxing

We use behavioral programming (BP) as defined in [8], as a formalism for defining the behavioral patterns and anti-patterns model. In the context of BP these patterns are called behavioral threads or b-threads. Since we want to explore possible future executions of the code, we describe mechanisms to run the code in a simulation mode (sandbox) and to retract the state when the search reaches the required depth. To this end, we use the Rhino (http://www.mozilla.org/rhino/) tool that allows to control the execution of JavaScript code from a host Java application. Specifically, the main feature that we use in Rhino is the ability to traverse the state-space of a given JavaScript program using continuations, as explained in more details below. In [15], continuations were used for model-checking behavioral programs. Unfortunately, this work relied on a Java package that was discontinued (not available for versions of Java beyond 1.6). Therefore, we developed an alternative tool using Rhino, called BPjs [33], and apply it, beyond offline model-checking, to online heuristic search.

Figure 1.1 illustrates the sandboxing architecture that we propose:

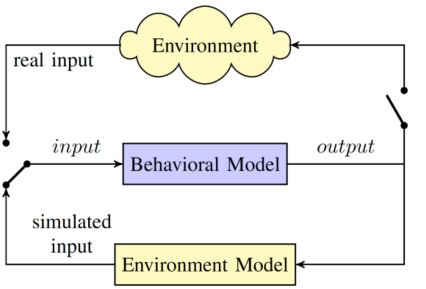


Fig. 1.1 The sandboxing architecture for interweaving AI and BP. During search, the environment events are simulated. When interacting with the real system, the simulation is updated.

We rely on a switched system that operates in two modes. (1) In normal mode: the application b-threads receive input from the environment. (2) In search mode: the application b-threads get input from a simulator and their output is fed only to the simulation b-threads, so that the environment is not actuated. The figure shows the state of the switches in search mode. When in normal mode, both switches are in their other mode, i.e., the right switch is closed and the left switch is connecting 'real input' to 'input'. The

box labeled "Environment Model" in this diagram represents code that programmers provide, which describes (an abstraction of) the environment in which the robot operates. While this may be considered as an additional burden on the programmers, we believe that it is a necessary artifact, even without our tool, as it is needed for testing and for verification.

Note that the simulation of the environment does not have to be complete: a good search mechanism should be able to make use even of an abstract description of the environment, as we will demonstrate in the case study described in the next section.

1.5 State Space Exploration

Programs in BP are comprised of b-threads and their states are an aggregate of the independent state of each b-thread. Anything happening inside a b-thread between b-sync calls — that is, between synchronization points — is by definition internal to the b-thread, and so can be considered atomic to the program as a whole. Therefore, our search mechanism can ignore these internal workings and focus on b-syncs, as only the states at these synchronization points define the integrated system behavior. As demonstrated with details in [33], this gives a significant reduction of complexity when compared to direct code step analysis as done, e.g., by the JavaPathfinder tool [4].

The programmatic construct that captures program execution in an immutable, re-entrant object, as required by search algorithms, is a continuation [34, 35]. Continuations are representations of the program at a given point in execution, which are available to the programmer, rather than hidden by the runtime environment. Our proposal here is to use continuations to traverse the state space by ordinary program execution, as they facilitate backtracking and resuming of execution from desired points where they were captured. We can now formally define the state space for the search using the terms proposed in [15]:

Definition: A bt-state represents the state of a specific b-thread in a b-sync call during a run of the program. It is composed of the b-thread and its captured continuation (program counter and values of all the variables). A bp-state is captured at every call to b-sync. This ensures that once all b-threads have reached a synchronization point, their continuation object representing that state is updated, so that a complete state of the BP system can be captured.

Definition: A bp-state represents the state of the whole program. It

is composed of the bt-states of all b-threads in the program, captured at a b-sync.

When the execution engine is required to make a choice between multiple events to trigger, it creates a bp-state as a root for the search. Expanding search nodes is done by triggering events and capturing the new bp-states created by the triggering. We propose to do this by executing the code as in [15] (and not by running on an abstract model of the code as, e.g., in [17]) to ensure that there are no discrepancies between the code and the search results. The bt-state and bp-state are the abstractions used by the search algorithm directly such that all BP specific code is encapsulated within those objects and is completely transparent to the search algorithm.

1.6 Heuristics and Smart Exploration

With the state space defined, we can now delve into the search mechanism itself: how we are running programs in a sandbox. The sandbox is composed of the environment simulator (input generator), a search algorithm, and a heuristic function. An implementation of a look-ahead mechanism, beyond one super-step, requires that the system be able to predict the actions of the environment to some precision. For this, we need to ask programmers to provide the search mechanism with an abstract model of the environment (which can be probabilistic), and a simulator, to provide inputs to the program while in the sandbox.

We implement the environment simulators as b-threads in the program itself. In normal operation, a simulator b-thread b-sync is modified such that its requested event sets are added to the waited-for events set, and its requested event set is empty. This ensures the b-threads simulation is made aware of all events relevant to it, so that it maintains a correct state for the next use in simulation mode. When the BP infrastructure needs to search for an event to trigger, the simulation b-threads are switched to simulation mode in which they request events normally triggered by the environment as modeled in their code. No manipulation of their event lists is done in simulation mode. This approach provides robustness of the simulator and program, while also allowing for a clear and unified programming interface.

An interface for sending inputs to the program and for examining its outputs is also required. A convenient solution for this is defining the interface to and from the program to be an event queue (similar to the input queue introduced in [36]). This way the environment and the sandbox both enqueue events for triggering within the program in the input queue and

read the program's output from its output queue. The program does not directly performs actions on the environment — events from the output queue are fed as actions back into the environment by an adapter, thus enabling running in sandbox without extra code analysis.

Selecting the right search algorithm for a program can have great impact on the results. The algorithms we have used in the work described below are depth-limited A* and minimax [37] textbook implementations [38]. The architecture of our solution is such that it is easy to introduce other search algorithms, as there is no coupling between the algorithm itself and the BP engine.

Our approach allows for any specification of heuristic functions: the bp-state object passed to the function grants access to the entire program without compromising speed or space. This includes the b-thread's b-sync event sets and public access methods. The programmer, then, is given full power in the evaluation of program state, independent of the search algorithm used. Our solution can incorporate domain-independent heuristics such as abstraction and pattern-databases [37]. Alternatively, the programmer can write different domain-specific heuristic functions that reward desired events and b-thread properties.

We propose to use an advanced search approach, based on the work of [39, 40]. Within combinatorial optimization, the term hyper-heuristics was first used in 2000 [41] to describe heuristics to choose heuristics. This definition of hyper-heuristics was expanded later [42] to refer to an automated methodology for selecting or generating heuristics to solve hard computational search problems. In the process of hyper-heuristics learning, heuristics are used as building blocks. These heuristics can be of high level, usually complex and memory-consuming (e.g., abstraction and pattern databases), or even low-level heuristics that are usually intuitive and straightforward to implement and compute.

HH-Evolver is a hyper-heuristic generator for search domains [40]. The HH-Evolver system receives as input: a domain, several heuristics for the domain, and a dataset of domain instances to be used partly as training set and partly as test set. HH-Evolver generates a population of random hyper-heuristics and trains them over generations against the training set. When used with a heuristic search algorithm, the individuals are required to produce near-optimal solutions to the instances encountered. HH-Evolver also seeks to reduce the search-size, i.e., the number of nodes encountered during the search.

Traditional heuristic methods rely on a single high-level heuristic or a

combination (usually a maximization) of several high-level heuristics. Low-level heuristics are disregarded as their input is less informative than the former ones. While this is true, there are numerous examples (some of them are listed in [40]) that using hyper-heuristic techniques, and HH-Evolver in particular, allow to efficiently incorporate low-level heuristics into a high-level heuristic. Moreover, in some domains producing a high-level heuristic is a hard task while low level heuristics can be easily produced. Our idea in applying HH-Evolver in an online is by maintaining a population of hyper-heuristics. Each individual in the population incorporates high-level heuristics as well as low-level ones, and simple domain knowledge. Over the generations, the individuals learn the conditions (i.e., logic functions) regarding when to apply each heuristic, or combinations thereof.

Beyond state exploration, we demonstrate in the examples below also the utility of parameter-space exploration. In many cases, programmers can specify the actions that their robot is required to perform, but they cannot specify the exact parameters of those actions. To provide a simple solution to this goal, we developed a programming style where some of the parameters of certain behavioral threads are left open by the programmer, assuming that they can best be assigned by an automatic solver. Specifically, we showed that such parameters can be effectively resolved using reinforcement learning. Using the infrastructure for simulation and backtracking, it is possible to run the system many times with many values for the unknown parameters until the algorithm converges to values that best achieve the goals that the programmers specified for the system.

1.7 Examples

To better understand the concepts of our approach, we now turn to present three examples. We begin with an ongoing work on two domains, both demonstrate the concept of state-space exploration. Next, we present a work that demonstrates the concept of parameter-space exploration.

1.7.1 *StarCraft*

In the following paragraphs, we describe a case-study we are running in programming virtual robots (bot) in a game called StarCraft which is a multi-player strategy game where players manage an army with many units of different types. We programmed a small aspect of the game having to do with mineral harvesting, as follows.

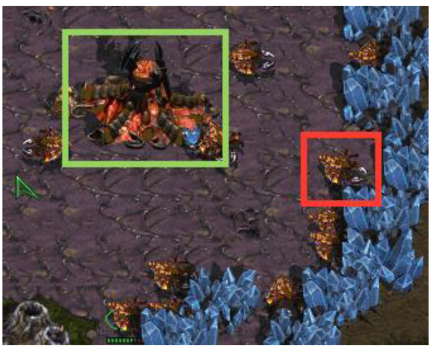


Fig. 1.2 A screenshot displaying our StarCraft example. The base is marked by the larger green square. The smaller red square marks a Zerg hatchery.

Figure 1.2 shows an example of using BP with search for optimal worker assignment to mineral fields in the game of StarCraft. We see a Zerg hatchery (marked by a small red square) with Drones (worker units, large green square) harvesting minerals (blue crystals). Drones collect and bring minerals to the hatchery, adding to the player's credit. The purpose of the bot is to assign drones to minerals in order to manage the harvesting process effectively. While it is possible to program the commands to the drones with standard programming languages, as many bot implementations do, our initial experiment here was to examine the benefits of providing only a partial implementation and to see how environment simulation can be done. Our program, then, specified only that some commands has to be given to each of the drones. The environment simulation specified roughly how the drones are expected to carry their commands. We used a very simple implementation of A* with hard-coded progress measure based on the number of collected minerals.

An initial comparison of our implementation with other implementations of this part of the bot logic shows that we were able to implement similar strategies but with much less code. The main reason for the slimness of our code is because we only had to specify the core of the behavior and the execution mechanism could adopt this core to the dynamic environment automatically. Indeed, much of the code in the other bot that we examined was about dealing with all kind of rare situations. This forms an initial indication that the direction that we are showing has merits.

1.7.2 *Robocode*

Robocode [43] is a programming game, where the goal is to develop a robot battle tank to battle against other tanks in Java or .NET (see Fig. 1.3). For many years Robocode has been used for education and research at schools and universities all over the world. The robot anatomy consists of three parts that are independently controlled: the body, that can move to all sides, and a gun and a radar, that can be rotated 360 degrees. Robocode has different leagues with different set of rules or presets (e.g., code-size limitations, one-on-one battles, battle royale, etc.), and users can test their robots against known robots of each league.

Fig. 1.3 A screenshot displaying the Robocode game. We see five tanks divided to four teams: blue, green, white and red. One of the green tanks is shooting at a red tank. One of the blue tanks is scanning frontwards with its radar. The names of the tanks are shown on the right.

The Robocode tutorial includes both basic and advanced robots, each contains a strategy or a combination of strategies for winning the game. For example, the *Tracker* is a robot that tracks and shoots the enemy, *Walls* is a robot that moves against the walls and targets the enemy using a targeting strategy called "head-on targeting", and so on. In addition, many strategies of top-rated robots are available as open source, allowing developers to test their robots against them. The plethora of strategies raises a problem — how to combine these strategies efficiently? The combination

must dynamically change according to the strategy of the enemies, their location, their status, etc. Moreover, deciding which strategy to advance at each phase has many considerations. Each action may advance more than one strategy, block others, achieve some sub-goals (e.g., detecting the enemy, dodging enemy fire, saving power, etc.) and block others. While experienced robot designers may have some intuitions regarding how to do it, writing them and integrating them with the code is hard and non intuitive.

One approach for dealing with this complexity, is to simply ignore such intuitions and domain knowledge, and utilize AI for learning new strategies. Evolutionary algorithms for example, have been used for learning many robots [44–46], each evolved with the main goal of outperforming known hand-written robots. Another goal, for example, is to win a specific category, e.g., [47] evolved robots for winning the HaikBot contest, which limits the code length of the robot. While some of these works achieved good results, integrating domain-expert knowledge with AI, may speed up the learning process, lead to dynamic adaption of the robots to the environment, and save time for the programmers, as they will only need to "spill out" intuitions, rather than explicitly define state-of-the-art policies.

In an ongoing work, we collected many strategies from different resources (i.e., tutorials and open-source robots), and specified each as a different set of b-threads. Next, we started "spilling out" the intuitions regarding which strategy to advance and added code that directs the execution engine using these intuitions. For that, we defined a set of sub-goals, as attributes that we wish to maximize. At each synchronization point of each strategy, we submitted a set of numbers between 0 to 1, each represents our intuition regarding how close we are to achieving a certain sub-goal. If we had no intuition — we submitted a `NULL` for that sub-goal. Finally, we used HH-Evolver [40] for learning how to prioritize these intuitions in respect to the current state of the game. We conducted two experiments so far, where we trained our robot against a single robot in a one-on-one competition. In the first experiment, we trained our robot against a "sitting-duck" robot, that only fires ahead. Of course, our robot quickly learned how to destroy its enemy. Next, we trained our robot against the *crazy* robot — a robot that constantly moves and fires in a random manner. Surprisingly, this robot is rated high among the sample robots [48]. Although the learning rate here was a bit longer, the evolved robot managed to develop a strategy for dodging the enemy's fire and destroying it. A video of these experiments can be viewed at `https://www.youtube.com/watch?v=kcgS1BSooBE`.

1.7.3 *RoboSoccer*

The following example was first presented in [49], where we demonstrated how our extension can be used to develop a controller for a robot in a virtual game called RoboSoccer (see Fig. 1.4). In this example we enhance the smart-execution mechanism using reinforcement learning, without using the sandboxing technique.

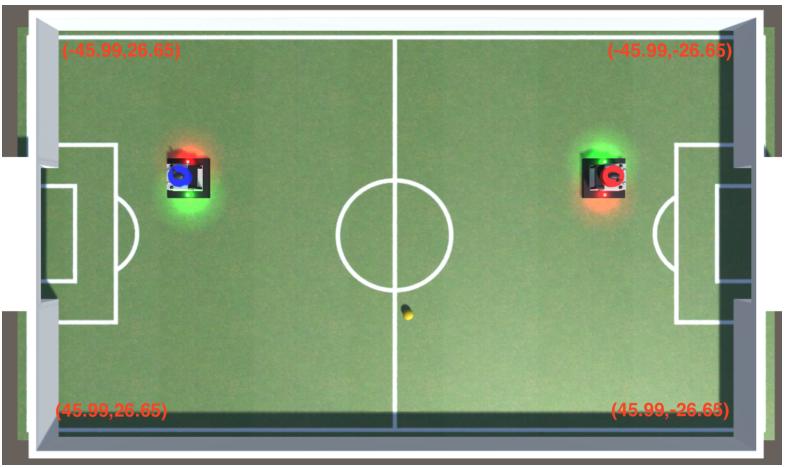


Fig. 1.4 A screenshot displaying our RoboSoccer example. Our model includes two b-threads for driving the robot towards the ball and for spinning it to the ball. A reinforcement learning mechanism handles the parameters tuning.

A simple model for driving the player (the red robot on the right) to the ball is given in Listing 1.1. This is a very simple BP model that does not integrate AI and consists of two b-threads, called move_towards_ball and spin_to_ball .

The b-threads start by waiting for a state update of the game and acting if the state matches the ball_is_free condition. The move_towards_ball b-thread takes care of controlling the speed of approaching the ball. If the distance is such that a gradient approach is needed, it requests a move_forward event with a gradient computed by the gradient function. Otherwise, it moves forward with full power. The spin_to_ball b-thread is responsible for controlling the direction that the robot moves to by spinning to the ball.

We note, that we used predefined constants in the code, such as TOO_CLOSE and TOO_FAR, that we calculated using a manual trial-and-error. We will now show how we use a reinforcement learning (RL) mechanism for this task. This setting allows the instructions to be simpler,

```
def move_towards_ball():
  while True:
    m = yield {}
    if not ball_is_free(m):
      continue
    dst = distance_from_player_to_ball(m);
    if dst < TOO_CLOSE:
      if dst < TOO_FAR:
        m = yield {request: move_forward(gradient(dst))}
      else:
        m = yield {request: move_forward(MAX_POWER)}
    else:
      if dst > (2 * TOO_CLOSE - TOO_FAR):
        m = yield {request: move_forward(gradient(dst))}
      else:
        m = yield {request: move_forward(-MAX_POWER)}

def spin_to_ball():
  while True:
    m = yield {}
    if not ball_is_free(m):
      continue
    deg = degree_from_player_to_ball(m)
    if deg > MAX_DEGREE:
      m = yield {request: spin(MAX_SPIN)}
    elif deg < -MAX_DEGREE:
      m = yield {request: spin(-MAX_SPIN)}
    else
      m = yield {request: spin(0)}
```

more robust, and easier to maintain when the simulation changes. The idea of merging BP with RL was first presented in [50]. Here, we show that with a combination of a rich solver and deep networks, the approach can also be used with multidimensional events and actions that contain numerical fields.

RL is a computational approach for understanding and for automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment [51]. In the standard RL model, an agent is connected to its environment via perception and action, as depicted in Fig. 1.5. On each step t of an interaction, the agent receives as input some indication of the current state, s_t, of the environment. The agent then chooses an action, a_t, to generate as output. The action

changes the state of the environment, and the value of this state transition is communicated to the agent through a reward signal, r_t. The goal is to find a *policy*, which is a function that gives the best action an agent can take, given the state of the environment, such that the long term reward is maximized.

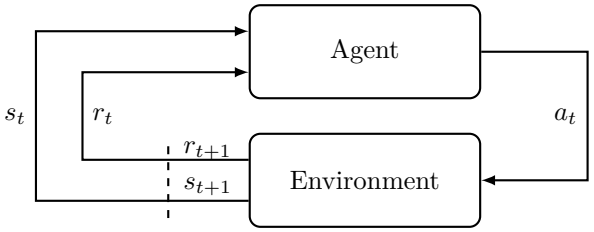


Fig. 1.5 A standard RL model. An agent is connected to its environment via perception and action.

In our setting, depicted in Fig. 1.6, the agent interacts with the b-program, that encapsulates the environment, in our case — the RoboSoccer game. The current state of the environment, s_t, consists of the player's position, compass, suction, and the ball's position. The action chosen by the agent, a_t, is used by the b-threads in order to modify the robot's behaviour. In this challenge, we targeted the task of grabbing the ball. As shown in Listing 1.2, the reward in each step, r_t, is defined by the get_ball_reward b-thread. This general model allows high modelling flexibility and generality, by allowing a model to be applied to various RL algorithms with different parameters of the environment.

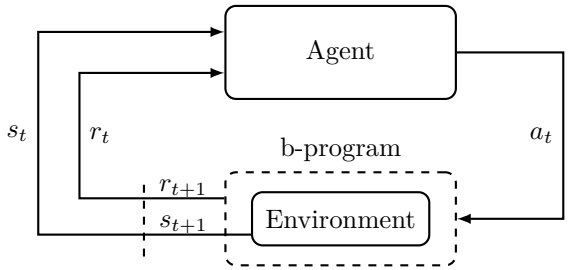


Fig. 1.6 A modified behavioral RL model. The environment is wrapped with a b-program.

In order to assist with the task of ball grabbing, the b-threads that

```
def get_ball_reward():
  m = yield {block: reward != -0.01}
  while True:
    if is_ball_in_robot(m):
      m = yield {block: reward != 1}
    else:
      m = yield {block: reward != -0.01}
```

are related for this task were simplified, by removing the exact conditions and parameters for the different actions. Instead, the activation of the actions now depends on the agent's commands. For example, in the modified move_towards_ball presented in Listing 1.3, the following parameters for the forward variable are controlled by the agent: half_speed_forward, full_speed_forward, half_speed_backwards, and full_speed_backwards. The spin_to_ball b-thread, which controls the direction of the spin, was modified in the same manner, as shown in Listing 1.4. Note that our goal is not to provide an interface to RL, but rather to provide programmers with a natural programming and modeling tool that applies RL under the hood. The key takeaway here is that we demonstrate how an intelligent execution mechanism can interpret more abstract commands that allow programmers to better break their models into modules that are aligned with the behavioral aspects that they perceive.

```
def move_towards_ball():
 m = yield {}
 while True:
   if not is_ball_in_robot(m):
     if half_speed_forward:
       m = yield {request: forward == MAX_PWR/2}
     if full_speed_forward:
       m = yield {request: forward == MAX_PWR}
     if half_speed_backwards:
       m = yield {request: forward == -MAX_PWR/2}
     if full_speed_backwards:
       m = yield {request: forward == -MAX_PWR}
    else:
       m = yield {}
```

Learning to control agents directly from high-dimensional sensory inputs, like the simplified RoboCup-type simulation state, is one of the long-standing challenges of reinforcement learning. Recent advances in deep

```
def spin_to_ball():
  m = yield {}
  while True:
    if not is_ball_in_robot(m):
      ang = angle_between_robot_and_ball(m)
      if need_to_spin and ang > 0:
        m = yield {request: spin > 0}
      elif need_to_spin and ang < 0:
        m = yield {request: spin < 0}
      else:
        m = yield {request: spin == 0}
    else:
       m = yield {}
```

learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in various domains [52]. In order for these techniques to be beneficial for RL with sensory data, reinforcement-learning approaches are augmented with deep neural networks (DRL). One of the most successful DRL algorithms is *Deep Q Network* (DQN) [52]. In its raw form, DQN uses a multilayer perceptron network for the policy function approximation.

In our implementation, which uses the DQN implementation of [53], the simulation state is being fed into the network as input. The output action of the DQN is then used by the move_towards_ball and spin_to_ball b-threads in order to modify the game controller. Note that we are not just using DRL to achieve automatic generation of a control strategy, our goal in this work is to simplify the software-engineering practices for robots software design. The end result of the example we have experimented with, is that the programmer could only specify modes (half_speed_forward, full_speed_forward, half_speed_backwards, and full_speed_backwards) and have the execution engine decide automatically when to activate each of them (based on a training session). Notice that this example shows how a constraint solver, which allows rich events in BP, and DRL, which allows learning from rich data, can complement each other.

1.8 Conclusions

In this chapter we outlined our approach for resolving the nondeterminism in scenario-based specifications by running reinforcement learning and online evolutionary algorithms that traverse possible continuations of execution. Using our mechanisms, programmers and other stakeholders can

provide reactive systems with programs (scenario-based specifications) that specify most of the behavior while leaving enough choice for the execution mechanism to cope with dynamicity in the environment.

References

[1] A. Elyasaf, A. Marron, A. Sturm and G. Weiss, A Context-Based Behavioral Language for IoT, in Regina Hebig and Thorsten Berger (ed.), *CEUR Workshop Proceedings*, Vol. 2245. CEUR-WS.org, Copenhagen, Denmark, pp. 485–494 (2018), `http://ceur-ws.org/Vol-2245`.

[2] M. Bar-Sinai, A. Elyasaf, A. Sadon and G. Weiss, A Scenario Based On-Board Software and Testing Environment for Satellites, in *59th Israel Annual Conference on Aerospace Sciences, IACAS 2019*, Vol. 2, ISBN 978-1-5108-8278-2, pp. 1407–1419 (2019), ISBN 978-1-5108-8278-2.

[3] H. Lapid, A. Marron, S. Szekely and D. Harel, Using Reactive-System Modeling Techniques to Create Executable Models of Biochemical Pathways, in S. Hammoudi, L. F. Pires and B. Selic (eds.), *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20–22, 2019*. SCITEPRESS - Science and Technology Publications, pp. 454–464 (2019), doi:10.5220/0007572504560466.

[4] G. Lindstrom, P. C. Mehlitz and W. Visser, Model Checking Real Time Java Using Java PathFinder, in D. A. Peled and Y. Tsay (eds.), *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4–7, 2005, Proceedings*, *Lecture Notes in Computer Science*, Vol. 3707. Springer, pp. 444–456 (2005), doi:10.1007/11562948_33, `https://doi.org/10.1007/11562948_33`.

[5] D. Harel and A. Pnueli, On the Development of Reactive Systems, in *Logics and models of concurrent systems*. Springer, pp. 477–498 (1985), doi:10.1007/978-3-642-82453-1_17.

[6] W. Damm and D. Harel, LSCs: Breathing Life into Message Sequence Charts, *Formal Methods in System Design* **19**, 1, pp. 45–80 (2001), doi:10.1023/A:1011227529550.

[7] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer Science & Business Media (2003), ISBN 3-540-00787-3.

[8] D. Harel, A. Marron and G. Weiss, Programming Coordinated Behavior in Java, in *ECOOP 2010 – Object-Oriented Programming*. Springer Berlin Heidelberg, pp. 250–274 (2010), doi:10.1007/978-3-642-14107-2_12.

[9] G. Wiener, G. Weiss and A. Marron, Coordinating and Visualizing Independent Behaviors in Erlang, in *Proc. 9th ACM SIGPLAN Work. Erlang.* ACM, pp. 13–22 (2010).

[10] B. Shimony, I. Nikolaidis, P. Gburzynski and E. Stroulia, On Coordination Tools in the PicOS Tuples System, in *Proceedings of the 2nd Workshop on*

Software Engineering for Sensor Network Applications. ACM Press, pp. 19–24 (2011).

[11] D. Harel, S. Maoz, S. Szekely and D. Barkan, PlayGo: Towards a Comprehensive Tool for Scenario Based Programming, in *ASE'10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ISBN 978-1-4503-0116-9, pp. 359–360 (2010), ISBN 978-1-4503-0116-9, doi:10.1145/1858996.1859075.

[12] H. Kugler, C. Plock and A. Roberts, Synthesizing Biological Theories, in *CAV*, pp. 579–584 (2011), doi:10.1007/978-3-642-22110-1_46, `http://link.springer.com/chapter/10.1007/978-3-642-22110-1_46`.

[13] D. Harel, A. Marron and G. Weiss, Behavioral Programming, *Communications of the ACM* **55**, 7, p. 90 (2012), doi:10.1145/2209249.2209270, `http://dl.acm.org/citation.cfm?doid=2209249.2209270`.

[14] D. Harel, H. Kugler, R. Marelly and A. Pnueli, Smart Play-out of Behavioral Requirements, in M. Aagaard and J. W. O'Leary (eds.), *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6–8, 2002, Proceedings*, *Lecture Notes in Computer Science*, Vol. 2517. Springer, pp. 378–398 (2002), doi:10.1007/3-540-36126-X_23, `https://doi.org/10.1007/3-540-36126-X_23`.

[15] D. Harel, R. Lampert, A. Marron and G. Weiss, Model-Checking Behavioral Programs, *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pp. 279–288 (2011).

[16] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi and G. Weiss, On Composing and Proving the Correctness of Reactive Behavior, in *Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, pp. 1–10 (2013), doi:10.1109/emsoft.2013.6658591.

[17] S. Maoz and Y. Sa'ar, Counter Play-Out: Executing Unrealizable Scenario-Based Specifications, in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, pp. 242–251 (2013), doi:10.1109/icse.2013.6606570.

[18] D. Harel, G. Katz, A. Marron and G. Weiss, Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs, *Transactions on Computational*, pp. 1–33 (2014), doi:10.1007/978-3-662-45896-9_1.

[19] D. Harel, A. Kantor and G. Katz, Relaxing Synchronization Constraints in Behavioral Programs, in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, pp. 355–372 (2013), doi:10.1007/978-3-642-45221-5_25.

[20] M. Brambilla, J. Cabot and M. Wimmer, Model-Driven Software Engineering in Practice: Second Edition, *Synthesis Lectures on Software Engineering* **3**, 1, pp. 1–207 (2017), doi:10.2200/s00751ed2v01y201701swe004.

[21] S. Kent, Model Driven Engineering, in M. J. Butler, L. Petre and K. Sere (eds.), *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15–18, 2002, Proceedings*, *Lecture Notes in Computer Science*, Vol. 2335. Springer, pp. 286–298 (2002), doi:10.1007/3-540-47884-1_16, `https://doi.org/10.1007/3-540-47884-1_16`.

[22] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, The Addison-Wesley object technology series. Addison-Wesley-Longman, Reading, Mass (2000), ISBN 978-0-201-65783-8.

[23] L. Balmelli, The Systems Modeling Language for Products and Systems Development, *The Journal of Object Technology* **6**, 6, p. 149 (2007), doi: 10.5381/jot.2007.6.6.a5.

[24] D. Hanselman and B. Littlefield, *Mastering Matlab 6: A Comprehensive Tutorial and Reference*. Pearson (2001).

[25] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis and P. Niebert, From Simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications, in F. Mueller and U. Kremer (eds.), *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11–13, 2003*. ACM, pp. 153–162 (2003), doi:10.1145/780732.780754.

[26] C. Ptolemaeus, *System Design, Modeling, and Simulation: Using Ptolemy II*, Vol. 1. Ptolemy. org Berkeley (2014).

[27] D. Steinberg, F. Budinsky, E. Merks and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education (2008).

[28] M. Harman, S. A. Mansouri and Y. Zhang, Search-based Software Engineering: Trends, Techniques and Applications, *ACM Comput. Surv.* **45**, 1, pp. 11:1–11:61 (2012), doi:10.1145/2379776.2379787.

[29] M. Sipper, *Machine Nature: The Coming Age Of Bio-Inspired Computing*. Tata Mcgraw-Hill Publishing Company Limited, New York (2002), ISBN 78-0071387040, `https://books.google.co.il/books?id=L_JFPgAACAAJ`.

[30] K. Kannappan, L. Spector, M. Sipper, T. Helmuth, W. G. L. Cava, J. Wisdom and O. Bernstein, Analyzing a Decade of Human-Competitive ("HUMIE") Winners: What Can We Learn? in R. L. Riolo, W. P. Worzel and M. E. Kotanchek (eds.), *Genetic Programming Theory and Practice XII, [GPTP 2014, University of Michigan, Ann Arbor, USA, May 8–10, 2014]*. Springer, pp. 149–166 (2014), doi:10.1007/978-3-319-16030-6_9, `https://doi.org/10.1007/978-3-319-16030-6_9`.

[31] D. Harel and I. Segall, Planned and Traversable Play-out: A Flexible Method for Executing Scenario-based Programs, in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07. Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-540-71208-4, pp. 485–499 (2007), ISBN 978-3-540-71208-4, `http://dl.acm.org/citation.cfm?id=1763507.1763556`.

[32] M. Georgeff, *Reasoning about Actions & Plans*. M. Kaufmann Publishers, Los Altos, Calif (1987), ISBN 978-0-934613-30-9.

[33] M. Bar-Sinai, *Extending Behavioral Programming for Model-Driven Engineering*, PhD Thesis, Ben-Gurion University of the Negev, Israel (2020).

[34] A. Assaf, A. Díaz-Caro, S. Perdrix, C. Tasson and B. Valiron, Call-by-value, Call-by-name and the Vectorial Behaviour of the Algebraic λ-calculus, *Logical Methods in Computer Science* **10**, 4 (2014), doi:10.2168/LMCS-10(4:8)2014.

[35] J. C. Reynolds, The Discoveries of Continuations, *Lisp and Symbolic Computation* **6**, 3–4, pp. 233–248 (1993).

[36] D. Harel, A. Kantor, G. Katz, A. Marron, G. Weiss and G. Wiener, Towards Behavioral Programming in Distributed Architectures, *Science of Computer Programming* **98**, pp. 233–267 (2015), doi:10.1016/j.scico.2014.03.003.

[37] S. Russell and N. Peter, *Artificial Intelligence: A Modern Approach*, 4th edn. Pearson Education (US) (2020), ISBN 0134610997, https://www.ebook.de/de/product/33612120/stuart_russell_peter_norvig_artificial_intelligence.html.

[38] S. Russell, Java Implementation of Algorithms from Russell and Norvig's Artificial Intelligence - a Modern Approach 3rd Edition. (2010), https://github.com/aimacode/aima-java.

[39] A. Elyasaf, A. Hauptman and M. Sipper, Evolutionary Design of Freecell Solvers, *IEEE Transactions on Computational Intelligence and AI in Games* **4**, 4, pp. 270–281 (2012), doi:10.1109/TCIAIG.2012.2210423, http://gateway.webofknowledge.com/gateway/Gateway.cgi?GWVersion=2&SrcAuth=ORCID&SrcApp=OrcidOrg&DestLinkType=FullRecord&DestApp=WOS_CPL&KeyUT=WOS:000312561100003&KeyUID=WOS:000312561100003http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6249736

[40] A. Elyasaf and M. Sipper, HH-Evolver: A System for Domain-Specific, Hyper-Heuristic Evolution, in *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference Companion*. ACM Press, New York, New York, USA, ISBN 978-1-4503-1964-5, pp. 1285–1291 (2013), ISBN 978-1-4503-1964-5, doi:10.1145/2464576.2482707, http://dl.acm.org/citation.cfm?doid=2464576.2482707.

[41] P. I. Cowling, G. Kendall and E. Soubeiga, A Hyperheuristic Approach to Scheduling a Sales Summit, in E. K. Burke and W. Erben (eds.), *Practice and Theory of Automated Timetabling III, Third International Conference, PATAT 2000, Konstanz, Germany, August 16–18, 2000, Selected Papers*, *Lecture Notes in Computer Science*, Vol. 2079. Springer, pp. 176–190 (2000), doi:10.1007/3-540-44629-X_11, https://doi.org/10.1007/3-540-44629-X_11.

[42] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan and J. R. Woodward, *A Classification of Hyper-Heuristic Approaches*, *International Series in Operations Research & Management Science*, Vol. 146, chap. 15. Springer, Boston, MA, ISBN 978-1-4419-1665-5, pp. 449–468 (2010), ISBN 978-1-4419-1665-5, doi:10.1007/978-1-4419-1665-5_15, https://doi.org/10.1007/978-1-4419-1665-5_15.

[43] F. N. Larsen, ReadMe for Robocode, *Retrieved February* **27**, p. 2015 (2013).

[44] J. Eisenstein, Evolving Robocode Tank Fighters, Tech. Rep., Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory (2003).

[45] D. Wyatt and D. Klein, Genetic Programming for Robocode Strategy, Tech. Rep., University of Washington (2003), https://courses.cs.washington.edu/courses/cse573/03au/reports/team-dan.pdf.

[46] R. Harper, Evolving Robocode Tanks for Evo Robocode, *Genetic Programming and Evolvable Machines* **15**, 4, pp. 403–431 (2014), doi:10.1007/s10710-014-9224-2.

[47] Y. Shichel, E. Ziserman and M. Sipper, GP-Robocode: Using Genetic Programming to Evolve Robocode Players, in M. Keijzer, A. Tettamanzi,

P. Collet, J. I. van Hemert and M. Tomassini (eds.), *Genetic Programming, 8th European Conference, EuroGP2005, Lausanne, Switzerland, March 30–April 1, 2005, Proceedings*, *Lecture Notes in Computer Science*, Vol. 3447. Springer, pp. 143–154 (2005), doi:10.1007/978-3-540-31989-4_13, `https://doi.org/10.1007/978-3-540-31989-4_13`.

[48] M. Nelson and F. N. Larsen, The Crazy Robot, (2017), `https://robowiki.net/wiki/Crazy`.

[49] A. Elyasaf, A. Sadon, G. Weiss and T. Yaacov, Using Behavioral Programming with Solver, Context, and Deep Reinforcement Learning for Playing a Simplified RoboCup-Type Game, in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, pp. 243–251 (2019), doi:10.1109/models-c.2019.00039.

[50] N. Eitan and D. Harel, Adaptive Behavioral Programming, in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, IEEE. pp. 685–692 (2011), doi:10.1109/ictai.2011.109.

[51] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Inroduction*. The MIT Press (2018), ISBN 9780262039246, `https://www.ebook.de/de/product/32966850/richard_s_sutton_andrew_g_barto_reinforcement_learning.html`.

[52] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, Playing Atari with Deep Reinforcement Learning, (2013), `arXiv:1312.5602 [cs.LG]`.

[53] A. Hill, A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor and Y. Wu, Stable Baselines: A Set of Improved Implementations of Reinforcement Learning Algorithms Based on OpenAI, (2018), `https://github.com/hill-a/stable-baselines`.

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0002

Chapter 2

AI Techniques for Software Requirements Prioritization

Alexander Felfernig

Institute of Software Technology
Graz University of Technology, Austria

2.1 Introduction

Limited resources, market demands, and technical restrictions regarding the implementation of software features often demand for the prioritization of requirements [1–4]. The focus of prioritization is the *ranking and selection of requirements that should be included in future software releases.* Intelligent decision support in prioritization is extremely important since especially when dealing with large assortments of requirements, manual prioritization processes tend to become very costly [5–8]. Potential suboptimal prioritizations can lead to different negative effects such as *waste of time* due to a focus on irrelevant requirements, *opportunity costs* due to the fact that the relevant features are not provided first, and *missing focus on market demands* that could lead in the worst case to total loss [9]. In this context, prioritization can take place on the strategic level as well as an on the operative level, which is typically associated with short-term prioritization tasks [10, 11]. The prioritization approaches discussed in this chapter are based on AI techniques from the areas of constraint reasoning & optimization [12], utility-based recommendation [13], content-based recommendation [14], matrix factorization [15], conflict detection [16], and model-based diagnosis [17].

An overview of different prioritization tasks is given in Fig. 2.1. This categorization is based on two dimensions. First, *level of requirements*

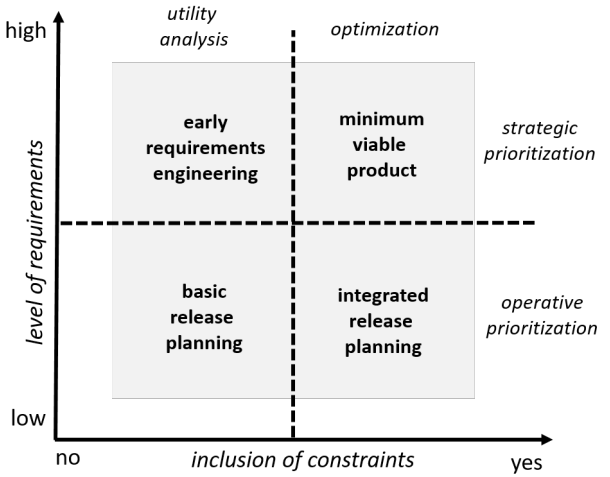


Fig. 2.1 Prioritization variants in software development contexts.

specifies the granularity of requirements specifications, i.e., to which extent these requirements can already be translated into corresponding detailed software features. Second, *inclusion of constraints* refers to which extent relationships between requirements and relationships to external factors are taken into account in the prioritization process. Examples of constraints (dependencies) between requirements are x *requires* y (x must not be implemented before y) and x *excludes* y (only one of these requirements should be implemented). Examples of external factors are the available budget for a software project, available personnel resources, and specific preferences of stakeholders engaged in a software project. Along with these two dimensions, there exist different prioritization approaches, which can be differentiated with regard to the *granularity level of requirements* and the *degree of the inclusion of constraints*.

Early requirements engineering is related to the idea of figuring out the requirements that have the highest importance, for example, for the market or specific customer communities. Prioritization tasks typically refer to high-level requirements, furthermore, no specific constraints are included. The major focus is to figure out the most relevant features of a product with a market relevance. Requirements in such scenarios can be regarded as high-level, for example, *"the new e-learning software should include a motivation functionality that persuades students to intensively learn the course topics"* or *"the new e-learning software should support natural language based interaction mechanisms"*.

A *minimal viable product* (MVP) should include a minimal set of features that can be integrated as parts of a fully operable software offered to customers. MVPs are a typical approach to get to the market as soon as possible with the most relevant features of a software. In this context, constraints play an important role since the prioritization of requirements has to take into account constraints such as available personnel and budget resources. Requirements can also be regarded as high-level and constraints primarily refer to available budgets and personnel resources. Examples of constraints are *"motivation features of the new e-learning software should solely include the aspect of social influence"* and *"for the first version of the software, natural language interaction should support the answering of multiple-choice questions with single correct answers"*.

Basic release planning does not fully take into account further constraints such as available budget, personnel resources, and time restrictions regarding the implementation of requirements. This type of prioritization covers implementation scenarios where releases are planned on an operational level without taking into account in detail constraints regarding available personnel and budget resources as well as time limitations. Examples of requirements in such contexts are *"the basic scenario for a social influence based persuasion is the following ... the user interface implementation of this function should look like as follows ..."* or *"the basic scenario for supporting multiple choice questions in the context of natural language interactions is the following ... the user interface implementation can be sketched as follows ..."*. On a technical level, basic release planning can be performed using approaches similar to those used in the context of early requirements engineering.

Finally, *integrated release planning* represents a full-fledged release planning [18,19] on the basis of detailed constraints representing organizational data and rules. In this context, both, constraints regarding dependencies between requirements as well as constraints related to external factors are taken into account. Similar to basic release planning, requirements are defined with fine granularity. A major difference between basic release planning and integrated release planning is the availability of more detailed constraint information, for example, integrated release planning is able to take into account the individual availability of developers (in terms of engagement in other projects and presence or absence during specific time periods). Furthermore, dependencies between requirements can be taken into account on a formal level.

On the level of prioritization techniques, there are *two basic approaches* to support prioritization processes — see Achimugu *et al.* [1]. *First*, prioritization can be regarded as an *optimization task* where the objective is to identify a prioritization that takes into account the preferences of individual stakeholders and also helps to optimize the prioritization with regard to a set of predefined constraints [20]. On a technical level, optimization-based prioritization is often based on a hybrid approach where the identification and aggregation of stakeholder preferences is supported by utility analysis [21–23] and optimization is performed on the basis of constraint reasoning [12, 24].

Utility-based approaches focus on an analysis of the given requirements with regard to a set of *interest dimensions* and less on automated optimization. Different variants of this approach can be implemented, for example, a utility-based ranking can be extended with the concepts of liquid democracy [25]. Finally, social networks can be exploited as data sources for the identification of new requirements, which are regarded as relevant by the underlying social network [26]. In terms of the application of the mentioned prioritization techniques, early requirements engineering and basic release planning focus more on utility-based prioritization approaches whereas minimum viable product and integrated release planning focus on optimization-based prioritization approaches.

The major contributions of this chapter are the following. First, we provide an overview of existing techniques that help to improve the quality of prioritization processes in requirements engineering. Second, we show the application of these techniques in the context of working examples. Third, in order to stimulate further work in related fields, we discuss relevant issues for future work.

The remainder of this chapter is organized as follows. Sections 2.2–2.5 include a discussion of the application of AI techniques in the scenarios of *early requirements engineering*, *minimum viable products*, *basic release planning*, and *integrated release planning*. These sections include a description of the underlying scenarios and working examples. Section 2.6 provides insights how to support stakeholder selection, which is an important issue when it comes to the assignment of requirement validation tasks. Section 2.7 provides an overview of issues for future research. Finally, we conclude the chapter with Section 2.8.

2.2 Early Requirements Engineering

A basic means to support prioritization tasks in early requirements engineering is to perform a utility analysis of a given set of requirements. *Utility-based prioritization* is based on the concepts of *multi-attribute utility theory* [27] — different variants thereof are possible. First, individual requirements are evaluated with regard to *interest dimensions* (e.g., risk level of a requirement and the commercial relevance of a requirement). The utility of the requirement is then determined on the basis of the sum of interest dimension specific utility values. Interest dimensions can be associated with a weight, for example, *low risks are more important than high profits*. Utility-based prioritization can also be implemented on the basis of analytic hierarchy process (AHP) [28]. A major disadvantage of this approach is that requirements have to be evaluated pairwise which does not scale well when the number of requirements increases.

Interest dimensions, i.e., basic evaluation criteria for utility-based prioritization can differ depending on the underlying decision scenario. Examples of such interest dimensions in company-related software projects are *effort to implement a requirement*, *risk of not being able to implement a requirement*, and *business relevance of a requirement (profit)* [1]. In open source settings, the dimensions can be different since open source contributors have to decide individually on which requirement to work next. Examples of related interest dimensions could be *personal expertise of an open source developer* and *importance of a requirement for the community* [29].

Utility analysis supports stakeholders in the prioritization of requirements with regard to a set of interest dimensions $D = \{d_1, d_2, ..., d_n\}$. The underlying idea is that requirements are first analyzed by individual stakeholders (also denoted as users) — see Tables 2.1–2.2. Such decisions are often group decisions where stakeholders are in charge of prioritizing a set of requirements [30].

In this simplified example, *users* are in charge of evaluating the requirements $req_1..req_5$ with regard to the interest dimensions *business relevance* and *risk*. Thereafter, individual evaluations are aggregated to determine the utility of requirements. In this context, Formula 2.1 can be used to calculate the utility of a requirement with regard to a specific interest dimension d. Furthermore, Formula 2.2 is used to determine the overall utility of a requirement.

Table 2.1 Evaluation of the dimension *relevance* (high rating = high relevance).

	$user_1$	$user_2$	$user_3$	$user_4$
req_1	1	4	5	2
req_2	10	6	1	7
req_3	2	6	5	2
req_4	1	1	3	7
req_5	7	8	6	5

Table 2.2 Evaluation of the dimension *risk* (high rating = low risk).

	$user_1$	$user_2$	$user_3$	$user_4$
req_1	2	7	3	2
req_2	9	9	1	7
req_3	2	10	3	2
req_4	2	5	3	1
req_5	3	2	3	5

$$utilityreq(req, d) = \frac{\Sigma_{u \in Users} eval(req, d, u)}{|Users|} \tag{2.1}$$

$$utility(req) = \frac{\Sigma_{d \in Dims} utilityreq(req, d) \times weight(d)}{|Dims|} \tag{2.2}$$

The determined utilities are then encoded in a ranking (see Table 2.3).

Table 2.3 Prioritization of requirements $req_1..req_5$ with regard to the interest dimensions *relevance* (weight = 0.75) and *risk* (weight = 0.25).

requirement req_i	req_1	req_2	req_3	req_4	req_5
utility(req_i)	4.63	**5.75**	4.06	2.94	4.56
priority(req_i)	2	1	4	5	3

The presented approach to group-based multi attribute utility analysis [30] is based on the assumption that each stakeholder is able to provide feedback on each of the given requirements. This might not be possible for various reasons, for example, stakeholders are simply not available, i.e., do not have time or they might have issues in terms of missing knowledge needed to evaluate a requirement. In such cases, mechanisms are needed to be able to transfer votes in a flexible fashion. Such an approach to liquid-democracy based prioritization is introduced in [25]. The major difference

compared to the aforementioned approach is that individual stakeholders are allowed to vote more than once and to transfer their votes to other stakeholders.

An alternative approach to handle *missing values* in requirements evaluation is to apply machine learning concepts, which help to automatically complete a potentially sparse rating matrix [15]. The automatically determined requirements evaluations can then be proposed to stakeholders and can also serve as indicators of potential issues related to contradictory evaluations, which have to be resolved. Table 2.4 depicts a user-item matrix, which includes a couple of missing evaluations (denoted with "?").

Table 2.4 Association of users with requirements $req_1..req_5$.

relevance	$user_1$	$user_2$	$user_3$	$user_4$
req_1	?	?	5	?
req_2	10	?	1	?
req_3	?	6	?	2
req_4	?	?	3	?
req_5	?	?	?	5

Based on the information included in Table 2.4, we can perform so-called dimensionality reduction and describe the relationship between users and requirements in terms of two low-dimensional matrices U and R where the former describes the relationship between users and abstract dimensions (hidden features) (see Table 2.5) and the latter the relationship between items and abstract dimensions (see Table 2.6).

Table 2.5 User × interest dimension ($d_1..d_3$) affinity matrix U.

	$user_1$	$user_2$	$user_3$	$user_4$
d_1	3,652807135	1,251029912	0,148850849	1,870385191
d_2	2,406538532	1,830201936	1,766613942	0
d_3	0,053547355	0,176813763	1,86544824	0,002507298

Table 2.6 Requirement × interest dimension ($d_1..d_3$) affinity matrix R.

	d_1	d_2	d_3
req_1	0,318390415	0,359262854	2,305033956
req_2	2,527786478	0,3177104899	0,035500999
req_3	1,072394897	2,524779729	0,126403403
req_4	0,167185814	1,181561695	0,467019398
req_5	2,665424355	0,109392275	0,008631143

The table entries can be learned on the basis of a matrix factorization approach that is based on non-linear optimization. The optimization goal is to find values for the low-dimensional tables, which help to predict the missing table entries as good as possible. For a detailed discussion of matrix factorization techniques we refer to [15].

Similar to the description of the relationship between users and hidden features, we can describe the relationship between requirements and hidden features. The higher the value, the higher the corresponding affinity between users (requirements) and the corresponding hidden features. We want to emphasize that in the matrix factorization context features are *hidden*, i.e., it is not clear if and which hidden feature corresponds to a specific evaluation dimension (as discussed in the context of utility-based prioritization).

The two low-dimensional matrices U and R can now be used to calculate a prediction for an unspecified user $\times$ requirement pair denoted with "?" (see Table 2.4). By applying matrix multiplication, we can, for example, determine a prediction of the evaluation of requirement req_1 by $user_1$. The corresponding table entry results from the expression $0,318390415 \times 3,652807135 + 0,359262854 \times 2,406538532 + 2,305033956 \times 0,053547355$ which is $2,151027154$. Expecting predictions on a scale 0..10, the prediction for the evaluation of requirement req_1 by $user_1$ appears to be rather low.

2.3 Minimum Viable Products

Minimum viable products (MVPs) represent products (in our case software components) that include a minimum set of requirements applicable and of value for a customer. In the context of software development, MVP development is extremely important especially for start-up companies since resources are often extremely limited and there is only one chance to develop the right product for the customer community. Consequently, prioritization support is extremely important in such scenarios. MVP development is related to DevOps software processes which are characterized by extensive automation and continuous updates [31]. Such processes support a more in-depth customer integration into feedback and prioritization and — as a consequence — help to increase the quality of prioritization due to deeper insights into the progress of the project.

Prioritizations for *minimum viable products* typically have to deal with high-level requirements, which do not describe specific functionalities but rather generic features of the software. For these features, it should be

made clear which are the most relevant ones that can realistically be implemented. We can consider the task of selecting a subset of requirements to be included in a minimal viable product as a utility-based prioritization task where requirement *utilities* and *time estimates* are used as basic inputs in a follow-up process that focuses on optimizing the selection of a bundle of most relevant features (requirements). Thus, MVP-oriented prioritization supports a kind of triage process [32] where the most important and feasible requirements are implemented first.

Formula 2.3 restricts the available time resources, i.e., how much time is available to implement the new MVP features. In typical start-up scenarios, this would reflect a situation where, for example, four persons together can spend around one month to implement market-relevant features into an MVP. To make good use of the available time, resource planning can be used to calculate an optimal subset of requirements to be included ($included(req_i)$) in the MVP. An example of how to take into account time restrictions is shown in Formula 2.3.

$$time(req_1) \times included(req_1) + .. + time(req_n) \times included(rec_n) \leq maxtime \tag{2.3}$$

The overall optimization objective of this resource planning task is expressed with Formula 2.4. The utility of the selected requirements (requirements, which should be part of the MVP) should be maximized while taking into account additional restrictions (see Formula 2.3).

$$max \leftarrow utility(req_1) \times included(req_1) + .. + utility(req_n) \times included(rec_n) \tag{2.4}$$

Table 2.7 Selecting the most relevant requirements under given time conditions resulting in a maximum utility of 10.31 = utility(req_2)+utility(req_5).

requirement req_i	req_1	req_2	req_3	req_4	req_5
utility(req_i)	4.63	**5.75**	4.06	2.94	**4.56**
time(req_i)	3	4	4	3	5
selected	0	**1**	0	0	**1**

2.4 Basic Release Planning

Basic release planning follows a prioritization approach where requirements formulated on a fine-granular level are selected with regard to their relevance of being part of one of the next n releases — in the case of $n = 1$,

this scenario is also denoted as *next release problem*. In most of the cases, such scenarios do not need the support of a high-sophisticated release planning solution. Example reasons for choosing a lightweight process are the *unavailability of resource data* required by release planning tools (e.g., data about resources already occupied in projects) and *limited budgets and personnel resources* to purchase and support a heavy-weight release planning software and to integrate this software with resource-related data sources.

Basic release planning focuses on the prioritization of requirements formulated on a fine-granular level. Initially, this process is often performed on the basis of a utility analysis (see Section 2.2). On the basis of the results of a utility analysis, stakeholders can propose assignments of requirements to releases. If a company's software process follows a *next release* strategy, i.e., the planning horizon is the next release, the corresponding selection task is to figure out the most relevant requirements for the next release. Basic release planning typically does not take into account constraints regarding available resources — such constraints are taken into account informally.

Tools supporting basic release planning can help to repair inconsistencies in the stakeholders' preferences regarding the assignment of requirements to releases. A scenario in the context of basic release planning is the following (see Table 2.8). Stakeholders (users) define their individual preferences regarding the assignment of requirements to releases. Since stakeholders can do this remotely and are initially often not allowed to see the preferences of other stakeholders, conflicts regarding defined release assignment preferences can occur [33].

Table 2.8 Preferences of stakeholders with regard to release assignments.

	user_1	user_2	user_3	user_4
req_1	1	1	2	1
req_2	2	2	3	3
req_3	3	3	3	3
req_4	1	2	2	3
req_5	4	1	1	1

In this context, constraint-based optimization can be applied to minimize the need of preference change per user (see Formula 2.7). We assume the existence of variables $ureq_{ij}$ with the domain 1..4 representing the releases 1..4, for example, $ureq_{11} = 1$ indicates that $user_1$ prefers the assignment of req_1 to release 1. Furthermore, we assume the existence of variables $ureq'_{ij}$, which represent the solution space. The constraint

$ureqcount_{ij} = abs(ureq_{ij} - ureq'_{ij})$ indicates whether a user preference has to be adapted. Furthermore, we need to count the number of changes needed per user i (see Formula 2.5). The number of preference changes per user i is represented by variable chn_i (see Formula 2.5).

$$chn_i \leftarrow ureqcount_{i1} + .. + ureqcount_{in} \tag{2.5}$$

Furthermore, we want to assure *consensus*, i.e., each requirement j has to be assigned to exactly one release (see Formula 2.6).

$$ureq'_{1j} = .. = ureq'_{mj} \tag{2.6}$$

Given this knowledge, we can define an optimization problem with the overall goal to minimize the number of changed release assignments while at the same time being fair, i.e., it should not be the case that (in the worst case) all needed changes are affecting a single stakeholder. This criteria is represented by Formula 2.7. The underlying idea is that the pairwise distance between stakeholders in terms of the number of needed stakeholder-specific preference adaptations should be minimized.

$$min \leftarrow abs(chn_1 - chn_2) + .. + abs(chn_{n-1} - chn_n) \tag{2.7}$$

Formula 2.8 represents an alternative optimization function where the expected solution represents a tradeoff between *fairness* among stakeholders in terms of a fair share of individual changes of preferences and *minimality* in terms of the overall number of needed changes.

$$min \leftarrow (abs(chn_1 - chn_2) + .. + abs(chn_{n-1} - chn_n)) \times (chn_1 + .. + chn_n) \tag{2.8}$$

This kind of knowledge can be exploited by optimization features of constraint solvers such as Choco.[1]

2.5 Integrated Release Planning

On top of the concepts of basic release planning, integrated release planning has a strong focus on integrating additional constraints related to the dependency between requirements and constraints related to the availability of resources, limits of resource consumption, and the assignment of stakeholders to individual tasks. Integrated release planning requires detailed information about the assignment of employees to current projects and their availability. Furthermore, project-specific release plans have to be synchronized since employees can be assigned to multiple projects during

[1] choco-solver.org

the same time period. A special case are distributed project scenarios where a large project is conducted by different independent teams that work on some common features, which have to be taken into account in the release plans of the individual project partners.

Table 2.9 provides a representative overview of modeling concepts that can be used in the context of release planning. Requirements can be represented as basic components with associated properties represented as finite domain variables. For example, $req_1.rel$ denotes requirement req_1 with the associated release $req_1.rel$, which could be represented, for example, by the domain 1..3, i.e., the look-ahead factor for releases would be 3. Another example of a property which can be associated with a requirement req_i is $req_i.dur$, which denotes the time estimate for requirement req_i.

Table 2.9 Examples of basic constraints used for defining release planning tasks. In this context, req_i denotes a requirement, $req_i.rel$ denotes the corresponding release, and $req.dur$ denotes the estimated development time for a requirement.

Definition	Description
$req_i.rel = a$	req_i is assigned to release a
$req_i.rel < req_j.rel$	req_i must be implemented before req_j
$req_i.rel \leq req_j.rel$	req_j must not be implemented before req_i
$req_i.rel \neq req_j.rel$	req_i and req_j must have different releases
$req_i.rel \leq a$	implementation of req_i not after release a
$req_i.rel \geq a$	implementation of req_i not before release a
$req_i.rel = n \vee req_j.rel = n$	req_i or req_j not in release plan
$\neg(\lvert req_i.rel - req_j.rel\rvert > k)$	req_i and req_j must be implemented timely
$\lvert\{r \in R : r = rel\}\rvert \leq a$	not more than a requirements in release rel
$\Sigma_{r \in R \wedge r.rel = rel}(r.dur) \leq a$	not more than a hours bounded to rel

A simple example of the application of the modeling concepts shown in Table 2.9 is given in Tables 2.10–2.11. Table 2.10 includes dependencies between requirements that are considered correct and have to be taken into account, i.e., the constraints are so-called *hard constraints*. For example $req_1.rel < req_2.rel$ denotes the fact that the implementation of req_1 has to be completed before the implementation of req_2 can be started. Since these constraints are assumed to be taken into account, they have to be consistent, i.e., at least one solution should exist. Assuming a finite domain of 1..3 for each individual variable $req_i.rel$, a corresponding consistent variable assignment (solution) is $\{req_1.rel = 1, req_2.rel = 2, req_3.rel = 3, req_4.rel = 3, req_5.rel = 1\}$.

Please note that all constraint types shown in Table 2.8 can be either

Table 2.10 Example requirements and set D of corresponding dependencies. The domain of $req_i.rel$ is assumed to be 1..3.

	$\text{req}_1.rel$	$\text{req}_2.rel$	$\text{req}_3.rel$	$\text{req}_4.rel$	$\text{req}_5.rel$
$\text{req}_1.rel$	-	$<$	-	-	-
$\text{req}_2.rel$	-	-	$<$	-	$>$
$\text{req}_3.rel$	-	-	-	-	-
$\text{req}_4.rel$	-	-	-	-	$\neq$
$\text{req}_5.rel$	-	-	-	-	-

Table 2.11 Example set S of (inconsistent) stakeholder preferences.

	user_1	user_2	user_3	user_4
$\text{req}_1.rel$	$= 1$	$= 1$	≤ 2	$= 1$
$\text{req}_2.rel$	≥ 2	≥ 2	≥ 2	≥ 2
$\text{req}_3.rel$	≤ 2	≥ 2	$= 3$	≤ 3
$\text{req}_4.rel$	≥ 1	≥ 1	≥ 2	≥ 2
$\text{req}_5.rel$	≥ 2	$= 1$	$= 1$	≤ 2

represented as *hard constraints* or as *soft constraints* — in the context of our example, the entries of Table 2.10 are interpreted as hard constraints, those of Table 2.11 as soft constraints, i.e., stakeholder preferences that should be taken into account but could also be ignored in the case that not all stakeholder preferences could be taken into account. On the basis of the (hard) constraints shown in Table 2.10, stakeholders (users) can specify their individual preferences (see Table 2.11). For simplicity, we restrict the constraint type of user preferences to the form $req_i.rel = a$, $req_i.rel < a$, $req_i.rel > a$, $req_i.rel \leq a$, and $req_i.rel \geq a$.

The stakeholder preferences S in Table 2.11 are inconsistent. Detailed release planning can be regarded as an interactive process where stakeholders define their preferences and then try to establish consensus with regard to the final release plan. In the example shown in Table 2.11, the stakeholders have defined inconsistent preferences with regard to the requirements req_3 and req_5. More precisely, there is one set of conflicting preferences with regard to req_3 ($\{\{user_1 : (\leq 2), user_3 : (= 3)\}\}$) and two conflicting preferences with regard to req_5 ($\{\{user_1 : (\geq 2), user_2 : (= 1)\}, \{user_1 : (\geq 2), user_3 : (= 1)\}\}$). Combinations of preferences that induce an inconsistency are often denoted as conflict set [16, 17]. Conflict sets can be shown to stakeholders to indicate open issues and to stimulate discussions on how to resolve the existing inconsistencies. In our example, the inconsistent situation could be resolved if stakeholder $user_1$ would agree to change both

of his (her) preferences. If we take into account both, the constraints in D and the preferences in S, we can detect two singleton conflicts both induced by the preferences of $user_1$ ($\{\{user_1 : (\leq 2)\}, \{user_1 : (\geq 2)\}\}$).

2.6 Stakeholder Recommendation

An issue in different prioritization scenarios is to figure out who should be in charge of validating a specific requirement since (s)he has the expertise needed. The quality of stakeholder/requirement assignment can have enormous impacts on the quality of a prioritization since sub-optimal evaluations can lead to sub-optimal prioritizations. Specifically, missing expertise can lead to situations where, for example, requirements of high relevance are evaluated as less relevant and — as a consequence — are not considered as a potential candidate for early releases. A major issue is to identify stakeholders who have the expertise and thus can provide reasonable evaluations of requirements. As sketched in Formula 2.9, expertise estimation can be implemented on the basis of the similarity between requirements already evaluated by a stakeholder and a set of new requirements.

Stakeholder expertise can be modeled in various ways. In the following, we provide a basic example of how to exploit the concepts of content-based recommendation [14] to propose reasonable assignments of stakeholders to requirements. Table 2.12 contains a set of new requirements with a corresponding set of keywords, which have been extracted from the requirement description. For these requirements, we would like to figure out automatically, which stakeholder would be the best one to work on this requirement, for example, to evaluate the requirement. Furthermore, Table 2.13 shows a list of stakeholders (users) and a corresponding list of keywords extracted from requirements descriptions the stakeholder worked on in the past. In order to estimate which stakeholder should work on which requirement, we can apply the concepts of content-based recommendation [14]. We can calculate the similarity between the keywords describing a stakeholder (see Table 2.13) and the keywords describing a requirement (see Table 2.12). This can be achieved by applying Formula 2.9 which helps to determine the stakeholder $\times$ requirements similarity.

$$sim(user, req) = \frac{2 \times |keywords(user) \cap keywords(req)|}{keywords(user) \cup keywords(req)} \tag{2.9}$$

The result of this similarity evaluation is summarized in Table 2.14. For req_1 and req_5, users with an average similarity have been identified as

Table 2.12 Requirements and keywords extracted from their descriptions.

Requirements	Keywords
req_1	registration users
req_2	basic payment
req_3	credit card payment
req_4	optimize user portfolio
req_5	optimize database

Table 2.13 Stakeholders and keywords of requirements they have validated.

Stakeholders	Keywords
$user_1$	registration feature database connection
$user_2$	payment process
$user_3$	credit card interfaces
$user_4$	credit card portfolio optimize

Table 2.14 Content-based similarity between stakeholders and requirements.

	$user_1$	$user_2$	$user_3$	$user_4$
req_1	**0.4**	0	0	0
req_2	0	**0.66**	0	0
req_3	0	0.5	**1.0**	**0.8**
req_4	0	0	0	**0.8**
req_5	**0.4**	0	0	**0.4**

candidates for validating the requirements. A user with a stronger similarity could be found for req_2. Finally, there is a strong similarity between requirements req_3, req_4, and $user_4$. Overall, $user_4$ seems to have a high coverage with regard to the potential requirements assignments. Finally, $user_3$ has the highest expertise with regard to a single requirement (req_3).

2.7 Research Issues

Derivation of Preferences from Social Networks. In the discussed prioritization scenarios, preference elicitation is still a manual process. Especially in contexts where companies have established a social network representing their user community, network contents, for example, in the form of tweets can be exploited to infer new requirements and preferences with regard to existing and future software features [26]. The automated integration of community preferences into requirements prioritization is still an open

issue and extremely relevant for making related decision processes more community-oriented and efficient. Beyond automated preference integration, quality assurance for preferences is an extremely important issue. [34] show how a consequence-based evaluation of different choice alternatives can help to improve the overall quality of release planning decisions.

Avoidance of Decision Biases. Decision biases are related to shortcuts in decision making that can lead to sub-optimal decisions [35,36]. Being aware of such biases helps to improve the overall quality of decisions processes. An example of such a bias is *anchoring* where the item evaluations of one user that are already visible to other users who haven't evaluated the item up to now, can have an impact on the evaluation behavior of other users [33]. For an overview of decision biases in recommender systems we refer to [35]. Many of the existing biases reported in the psychological literature have not been evaluated up to now. This can be regarded as a major topic for future research.

Transparency of Decisions. In order to increase trust, decisions have to be made transparent. Transparency can be achieved on the basis of explanations, which help to understand the reasons for a recommended decision [30]. An important role of transparency is also related to the task of avoiding manipulations in decision making [37]. An example thereof is a situation where a user tries to adapt his/her rating in order to push his/her preferred alternatives (*push attack*). As discussed in Trang *et al.* [37], a very effective way of avoiding manipulations is to make the rating behavior of individual users more transparent, i.e., making their rating behavior visible to other users. A research issue in this context is to analyze in detail which degree of transparency of rating behavior best helps to counteract manipulations and which visualizations should be used to explain the current status of a decision process.

Prioritization and Decision Making in Open Source Environments. Open source development often takes place in the context of single user (contributor) decision making, i.e., contributors can individually and independently decide which requirement to implement next. Often, many new requirements are potential candidates and the analysis of these candidates is time-consuming. In this context, prioritization can help to automatically rank new requirements in a contributor-specific fashion and thus to significantly reduce related analysis efforts. An approach to support such prioritization scenarios in the ECLIPSE open source environment is reported in [29]. A research challenge in this context is to develop decision support approaches that do not only determine recommendations for individuals

but also to figure out which prioritization helps to make the open source community as a whole more productive.

2.8 Conclusions

In this chapter, we provide an overview of prioritization scenarios that can be differentiated with regard to the degree of underlying requirement granularity and whether constraints are used to describe a prioritization task. These scenarios range from *early requirements engineering* (utility analysis of high-level requirements), *minimum viable product* (selection of features to be contained in a first version of a product), *basic release planning* (initial prioritization of requirements), to *integrated release planning* (detailed prioritization of requirements with regard to a predefined set of releases). To better show the application of related decision support techniques, we introduce a couple of prioritization examples. This chapter is concluded with an outline of open issues for future research.

Acknowledgment

The work presented in this chapter has been conducted within the scope of the Horizon2020 OpenReq Project (funded by the European Union).

References

[1] P. Achimugu, A. Selamat, R. Ibrahim and M. Mahrin, A systematic literature review of software requirements prioritization research, *Information and Software Technology* **56**, 6, pp. 568–585 (2014).

[2] M. R. Karim and G. Ruhe, Bi-objective genetic search for release planning in support of themes, in *Proceedings Symposium on Search Based Software Engineering*. Springer, pp. 123–137 (2014).

[3] L. Lehtola, M. Kauppinen and S. Kujala, Requirements prioritization challenges in practice, in *5th International Conference On Product Focused Software Process Improvement (PROFES)*. Kansai Science City, Japan, pp. 497–508 (2004).

[4] B. Mobasher and J. Cleland-Huang, Recommender Systems in Requirements Engineering, *AI Magazine* **32**, 3, pp. 81–89 (2011).

[5] M. Alenezi and S. Banitaan, Bug reports prioritization: Which features and classifier to use? in *12th International Conference on Machine Learning and Applications*, pp. 112–116 (2013).

[6] A. Perini, F. Ricca and A. Susi, Tool-supported requirements prioritization: Comparing the AHP and CBRank methods, *Information and Software Technology* **51**, 6, pp. 1021–1032 (2009).

[7] G. Ruhe, Software engineering decision support–a new paradigm for learning software organizations, in *International Workshop on Learning Software Organizations*. Springer, pp. 104–113 (2002).
[8] J. Xuan, H. Jiang, Z. Ren and W. Zou, Developer prioritization in bug repositories, in *34th International Conference on Software Engineering (ICSE)*. Zürich, Switzerland, pp. 25–35 (2012).
[9] D. Firesmith, Prioritizing Requirements, *Journal of Object Technology* **3**, 8, pp. 35–47 (2004).
[10] D. Ameller, C. Farre, X. Franch, D. Valerio and A. Cassarino, Towards continuous software release planning, in *24th IEEE International Conference on Software Analysis, Evoluation and Reengineering (SANER)*, pp. 402–406 (2017).
[11] G. Ruhe and M. Saliu, The art and science of software release planning, *IEEE Software* **22**, 6, pp. 47–53 (2005).
[12] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, London (1993).
[13] A. Felfernig and R. Burke, Constraint-based recommender systems: Technologies and research issues, in *ACM International Conference on Electronic Commerce (ICEC08)*. Innsbruck, Austria, pp. 17–26 (2008).
[14] M. Pazzani and D. Billsus, Learning and revising user profiles: The identification of interesting web sites, *Machine Learning* **27**, pp. 313–331 (1997).
[15] Y. Koren, R. Bell and C. Volinsky, Matrix factorization techniques for recommender systems, *IEEE Computer* **42**, 8, pp. 30–37 (2009).
[16] U. Junker, QuickXplain: Preferred Explanations and Relaxations for Over-Constrained Problems, in *19th National Conference on AI (AAAI04)*. San Jose, CA, pp. 167–172 (2004).
[17] A. Felfernig, M. Schubert and C. Zehentner, An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing (AIEDAM)* **26**, 1, pp. 175–184 (2012).
[18] M. Nayebi and G. Ruhe, Analytical product release planning, in *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, pp. 550–580 (2015).
[19] G. Ruhe, *Product release planning: methods, tools and applications*. CRC Press (2010).
[20] F. Kifetew, A. Susi, D. Mutante, A. Perini, A. Siena and P. Busetta, Towards multi-decision-maker requirements prioritisation via multi-objective optimisation, in *Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering (CAiSE'17)*. Essen, Germany, pp. 137–144 (2017).
[21] G. Adomavicius, N. Manouselis and Y. Kwon, *Recommender Systems Handbook*, chap. Multi-Criteria Recommender Systems, 1st edn. Springer, pp. 769–803 (2010).
[22] S. Huang, Designing utility-based recommender systems for e-commerce: Evaluation of preference elicitation methods, *Electronic Commerce Research and Applications* **10**, 4, pp. 398–407 (2011).
[23] K. Wiegers, *Software Requirements*. Microsoft Press (2003).

[24] G. Ninaus, A. Felfernig, M. Stettinger, S. Reiterer, G. Leitner, L. Weninger and W. Schanil, Intellireq: Intelligent techniques for software requirements engineering, in *European Conference on Artificial Intelligence, Prestigious Applications of Intelligent Systems (PAIS)*, pp. 1161–1166 (2014).

[25] M. Atas, T. Tran, R. Samer, A. Felfernig and M. Stettinger, Liquid democracy in group-based configuration, in *Workshop on Configuration*. CEUR, Graz, Austria, pp. 93–98 (2018).

[26] G. Williams and A. Mahmoud, Mining twitter feeds for software user requirements, in *25th International Requirements Engineering Conference (RE)*. IEEE, Lisbon, Portugal, pp. 1–10 (2017).

[27] J. Dyer, Multi attribute utility theory, *International Series in Operations Research and Management Science* **78**, pp. 265–292 (1997).

[28] J. Karlsson and K. Ryan, A Cost-Value Approach for Prioritizing Requirements, *IEEE Software* **14**, 5, pp. 67–74 (1997).

[29] A. Felfernig, M. Stettinger, M. Atas, R. Samer, J. Nerlich, S. Scholz, J. Tiihonen and M. Raatikainen, Towards utility-based prioritization of requirements in open source environments, in *26th IEEE Conference on Requirements Engineering*. IEEE, Banff, Canada, pp. 406–411 (2018a).

[30] A. Felfernig, L. Boratto, M. Stettinger and M. Tkalcic, *Group Recommender Systems – An Introduction*. Springer (2018b).

[31] L. Lwakatare, T. Kilamo, T. Karvonen, T. Sauvola, V. Heikkiläc, J. Itkonen, P. Kuvaja, T. Mikkonen, M. Oivo and C. Lassenius, DevOps in practice: A multiple case study of five companies, *Information and Software Technology* **114**, pp. 217–230 (2019).

[32] A. Davis, The art of requirements triage, *IEEE Computer* **36**, 3, pp. 42–49 (2003).

[33] M. Stettinger, A. Felfernig, G. Leitner and S. Reiterer, Counteracting anchoring effects in group decision making, in *23rd Conference on User Modeling, Adaptation, and Personalization (UMAP'15)*, *LNCS*, Vol. 9146. Springer, Dublin, Ireland, pp. 118–130 (2015).

[34] M. Nayebi and G. Ruhe, Asymmetric release planning: Compromising satisfaction against dissatisfaction, *IEEE Transactions on Software Engineering* **45**, 9, pp. 839–857 (2018).

[35] A. Felfernig, Biases in decision making, in *Proceedings of the International Workshop on Decision Making and Recommender Systems 2014*, Vol. 1278. CEUR Proceedings, Bolzano, Italy, pp. 32–34 (2014).

[36] A. Felfernig, W. Maalej, M. Mandl, M. Schubert and F. Ricci, Recommendation and decision technologies for requirements engineering, in *ICSE 2010 Workshop on Recommender Systems in Software Engineering*. Cape Town, South Africa, pp. 1–5 (2010).

[37] T. Tran, A. Felfernig, V. Le, M. Atas, M. Stettinger and R. Samer, User interfaces for counteracting decision manipulation in group recommender systems, in *27th ACM Conference on User Modeling, Adaptation and Personalization (UMAP)*. Larnaca, Cyprus, pp. 93–98 (2019).

This page intentionally left blank

Agent-Based Software Programming

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0003

Chapter 3

Social Commitments for Engineering Interaction in Distributed Systems

Matteo Baldoni, Cristina Baroglio, Roberto Micalizio and Stefano Tedeschi

Università degli Studi di Torino Dipartimento di Informatica

3.1 Introduction

Multiagent Systems (MAS) [1] are an effective choice for the design and development of distributed systems that involve components which act independently (i.e., autonomously). The scenario, in fact, calls for the use of four major software engineering techniques to cope with size and complexity — namely, modularity, distribution, abstraction, and intelligence (i.e., flexibility) — and the MAS paradigm encompasses all of them [2].

Nowadays, agent-oriented software engineers can choose from a substantial number of agent platforms (see, e.g., [3] for an overview). Tools like JADE [4], TuCSoN [5], DESIRE [6], and JaCaMo [7] all provide coordination mechanisms and communication infrastructures. However, the limit of the best-established platforms is a lack of abstractions for explicitly modeling interaction as a first-class entity. All of them provide communication infrastructures, mainly as message passing, but none of them encompass a mechanism to explicitly represent and manipulate the relationships created by the agents during their interaction. The lack of such a mechanism impairs the agents to reason on how to get their goals by engaging with others. As noted in [8], MAS, by their nature, do not include a centralized control mechanism, and agents are expected to reason about what interactions to engage with others. This peculiarity endows agents with a significant flexibility of action, but the current platforms, instead of explicitly accounting for a "society" layer [9], for regulating and norming how agents can act and interact within the system, their constraints and the system laws, force the

projection of the regulations directly inside the behaviors of the agents. As a consequence, MAS infrastructures do not really preserve the agents' autonomy and they do not fit the high degree of decoupling which is expected of the agents. This choice overly ties agent implementation with a negative impact on software reuse, and also on the realization of open MAS, that is, MAS that agents, possibly heterogeneous and developed by independent parties, can dynamically join/leave. Thus, when interaction is hard-coded as an exchange of messages, the advantage of having autonomous agents is significantly reduced. Instead, if social relationships are represented as first-class entities, that is, resources that agents can manipulate, they also become synchronization tools, reducing coupling between agents and improving flexibility (for a detailed discussion see [10]).

As a first contribution of this chapter, we illustrate the advantages of including an explicit representation of the *social relationships* that tie the agents, and in particular the positive impact such a new explicit representation would have on code modularity and interaction flexibility. We practically demonstrate these advantages when social relationships are modeled as *social commitments* [11] and *reified* as resources in the environment. A social commitment is a promise that an agent (debtor) makes to another one (creditor) to bring about some condition of interest. It is, therefore, a relationship with a distributed nature, directed from the debtor to the creditor. It engenders rights on both sides. For example, the creditor has the right to complain if the commitment is not fulfilled and the debtor has the right to expect its performance of the action concerned to be accepted as the fulfilment of this commitment. So, social commitments have a normative power, yielding obligations and expectations: by withdrawing from a social commitment, an agent violates an obligation, and frustrates expectations and rights [8, 12].

In a system where interaction is ruled by a message-passing protocol, the parties are designed so as to be compliant to the protocol. The expectations that one party has on the others, as well as the corresponding obligations, are granted "extra-program", in the sense that they are yielded by the protocol (i.e., the standard) to which each of them *declares* to comply to. Social commitments capture such a feature by their own nature (indeed, the above declaration is, by itself, the commitment of the party to adhere to the protocol specification). Building upon this and following the direction postulated in [13], in this work we use them as building blocks for creating the interaction standards themselves — possibly in a dynamic way, depending on the agent goals and on the context. Social commitments, created by their debtors, are promises that such agents make to behave in a way that complies to the achievement of the expressed condition, resulting into an obligation for the involved agents. All the agents that can observe the commitment will now expect something specific to occur sooner or later. The advantage of

social commitments, w.r.t. message-passing protocols, is that they are created by the agents depending on their own objectives and decisions, this promotes some important software engineering properties as we discuss below.

As a second contribution, we show that when commitments are realized as resources, that can be manipulated directly by the agents, the system as whole can gains some important benefits. In particular, agents can use commitments to reason upon their interactions, and can determine how and when engaging with other agents to achieve goals of their own interest. This is a significant contribution for the MAS research area, where, despite one of the most important characteristics of the MAS paradigm being agent situatedness, most studies are focussed only on features of the agents. At the same time, those that put forward the need of representing the environment typically do not provide a representation of the process, by which data evolve, in a form that can be reasoned about. This hampers interaction because agents are unable to create expectations on how others will act upon data that are part of the environment. Proposals like [14, 15] introduce first-class abstractions for the environment, to be captured alongside agents themselves. In particular, [15] states that "the environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources." This proposal brought to the Agents & Artifacts (A&A) meta-model [16], and its implementation CArtAgO [17]. A&A enables the agents to mediate their interactions by means of shared computational resources — the artifacts — which can encapsulate information. Hence, they have the advantage of a clear separation between the agent deliberative cycles and the data upon which this cycle is carried out. However, artifacts do not encompass a normative dimension, and this prevents agents to create expectations about others.

A similar proposal has been put forward by [18] in the business process setting. They propose business artifacts as business-relevant objects that are created and evolve as they pass through business operations. Business artifacts include an information model of the data, and a lifecycle model. The latter captures the key states through which data evolve and their transitions, and it is used both at runtime (to track the evolution of business artifacts), and at design time (to distribute tasks among the processes that operate on a business artifact). Also business artifacts, however, do not provide any link to a corresponding normative understanding, thus making impossible for the agents (processes) to leverage this knowledge for reasoning about how to act.

Last but not the least, social commitments, and their normative dimension, allow one to define the agent programming patterns. The commitment lifecycle, in fact, provides a clear direction on how agents participating to an interaction

should be implemented. We present four programming patterns that relate the individual agent's goals with her role, as debtor or creditor, in a set of commitments. These patterns help a programmer implement agents' behaviors in the following cases: create a new commitment to foster cooperation from others (*entice*), take part into an interaction by moving a commitment towards its satisfaction (*cooperate*), withdraw an offer when it is no longer useful (*withdraw*), and release a commitment when the made offer is not appealing (*give up*). Notably, these patterns are independent of any specific agent platform because they only depend on the standardized lifecycle of commitments. We exemplify the use of the patterns for agent programming in the context of a realistic hiring scenario using the JaCaMo+ platform [19].

The chapter is organized as follows. In Sec. 3.2 we exploit a motivating example for positioning our contribution against the current approaches for dealing with goal distribution and coordination of a number of independent execution threads. Section 3.3 provides some background notions about commitments that are essential for the proposal we present in Sec. 3.4, and highlights its advantages from a software engineering perspective. Agent programming patterns driven by the commitment lifecycle are presented in Sec. 3.5, followed by an example about the use of these patterns in JaCaMo+ in Sec. 3.6.

3.2 Background on Goal Distribution and Coordination

Not always results can be achieved in isolation, but require an interaction with others. This happens, for instance, when an agent must rely on an action performed by another agent, for necessity or for convenience. However the two agents could likely have different goals, which then have to be accommodated during the interaction. The possibility to reach each actor's goal depends on the evolution of the interaction and, thus, an effective coordination is crucial [2].

Example 3.1 (Item Purchase). Let us consider, as an illustrative example, a purchase scenario, involving a customer willing to buy a given quantity of an item from a merchant. The scenario is inspired from the well-known Netbill protocol [20]. When the customer requests a quote for the items from the merchant, the latter must send the quote. If the customer accepts the quote, the merchant sends the items, then waits for an electronic payment. We assume that the items cannot be used without a decryption key, sent by the merchant after the payment together with a receipt.

Coordination in Business Processes. A possible way to model Example 3.1 is by representing the customer and the merchant as two interacting BPMN (Business Process Model and Notation) Processes [21], each one pursuing a different objective, as reported in Fig. 3.1. The progression of each process depends on the other one, and requires coordination. In particular, the ultimate goal of the merchant (i.e., to complete the purchase) depends on the reception of several messages from the customer. However, the two processes are "separate", even if interacting (e.g. they might have been developed within different companies). For this reason no assumption can be made on the actual behavior of the customer. This is the reason why the customer's process is not shown in Fig. 3.1.

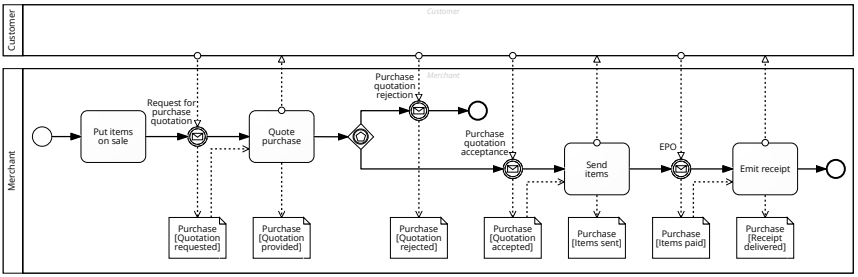


Fig. 3.1: BPMN diagram of the *Purchase* scenario.

A substantial limitation of this approach is that it does not capture well how the information relevant for the progression of the purchase evolves. Such information can be modeled as a *data object*, whose state evolves along with the execution of the process, in order to support the *data consistency*, i.e. that the data used in each step of the process does not contain contradictions w.r.t. the data used and produced in the previous steps. However, this is not sufficient because neither the process specification nor the data object specification include "causal" relationships between the actions of the various agents involved [22]. For instance, an agent may know that by paying an item the purchase will change state to "paid", but this is not enough to allow the agent to have expectations about the item being shipped as a consequence of paying. So, why paying? It is not a goal of the agent to make the purchase pass to the state "paid", but rather to have the item delivered. Paying is instrumental to delivery but delivery depends on other agents. What we need here is something that allows the buyer agent to legitimately have the expectation that the item will be delivered after paying. In business processes, the interaction between the two processes is loosely modeled in terms of message exchanges, thus not allowing each one to create expectations about other's behavior. When a business goal is split over a set of interacting processes, each of these

will realize only a part, and, in order to realize such a part, it will generally depend on the achievement of sub-goals that are realized by other processes. The synchronization realized in BPMN through message-passing or by the introduction of shared data storages does not capture coordination in high-level terms, that enable the agents, that dynamically join the system, to understand the obligations they can legitimately draw as consequences of their actions.

A more *data-oriented* approach is to model the purchase as a CMMN (Case Management Model and Notation) Case [23], as illustrated in Fig. 3.2. CMMN is a graphical notation used for capturing the handling of particular situations (i.e. *cases*) requiring various activities that may be performed in an unpredictable order in response to evolving situations. CMMN overcomes some of the limitations of BPMN, allowing to model work efforts which are less structured and depend on events, possibly occurring as a result of the evolution of data. A CMMN case is represented as a set of activities, possibly depending one on another, to be preformed in response to certain events. The activity execution makes the case evolve, running across multiple *milestones*. Despite allowing to encompass the evolution of data in the modeling of processes, CMMN still fails in explicitly representing the mutual engagements between the actors involved in a distributed processes.

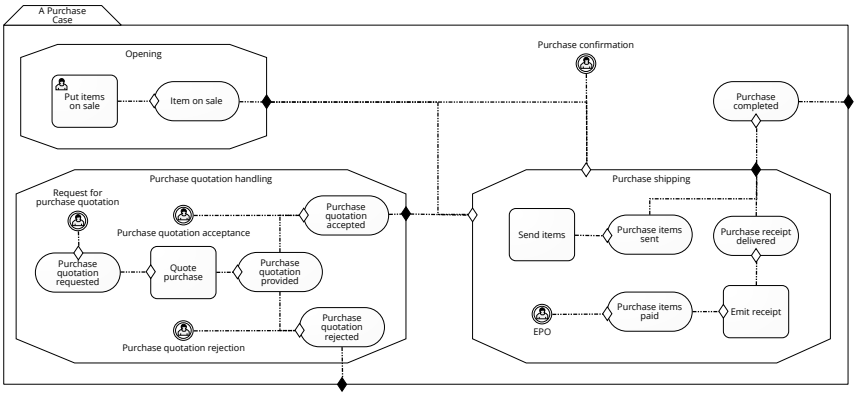


Fig. 3.2: CMMN diagram of the *Purchase* scenario.

Coordination with Business Artifacts. An alternative approach to cope with coordination is the adoption of *business artifacts*. Business artifacts add an information layer concerning both the structure and the lifecycle of the data they encompass. Some authors [24] propose to use them as a means to combine process

engineering with data engineering. Still, as explained in [25], they do not support coordination satisfactorily. To understand why, let us focus on the BALSA (Business Artifacts with Lifecycle, Services, and Associations) methodology [26], that we consider as a significant representative of the business artifacts approaches. Here, coordination among of business processes is tackled by relying on *choreographies*, regulating different business processes, which access to a same business artifact. Choreographies realize a form of *subjective coordination* [27]. This means that each business process needs to include also the "interaction logic" of the choreography role it plays along with its business logic. In other words, there is a strict coupling among the implementations of all the interacting processes. Consequently, the design and implementation become more complex, and the possibility of reusing the same process in different contexts is reduced.

Coordination in Multiagent Systems. Since the early proposals for MAS programming, organizations have been seen as metaphors for modularizing the code. Organizations, in fact, provide an overall abstraction of the task the agents have to achieve. In Gaia [28, 29], for instance, organizations are characterized by two features: a set of roles and a set of interactions among roles. Here interactions are seen as protocol definitions; where a protocol is "an institutionalized pattern", namely, a pattern that has been formally defined [29]. The pattern, thus, defines the rules (i.e., *norms*) through which an institutional reality, i.e. an interaction scope delimiting these norms, takes shape and evolves [30]. This institutional reality is the actual means of coordination of the agents: its constituent elements, the *institutional facts*, have a social meaning, by way of norms, that is shared and understood by all the agents participating to the interaction. Agents, thus, act so as to bring about those institutional facts that represent their goals or duties towards others. In other terms, when norms are explicitly represented and known by all the participants, it is possible to create *expectations* about the behavior of others in response to given messages, and this allows determining when and how to act. Indeed, coordination is all about expectations: an activity can fruitfully be carried out by many parties when there is a clear understanding on what each one should do and when. So one party will wait for the completion of the task by another party before starting its part. On the same grounds, the party who is first to act is confident that another party will continue from where it stopped.

Tools. The development of MAS relies upon a number of frameworks. JaCaMo [7] is a programming platform that integrates agents, environments and organizations. It is built on top of Jason [31], an agent programming language,

CArtAgO [32] for programming environments, and $\mathcal{M}$OISE [33] for programming organizations. 2COMM [10, 34] is a middleware that provides functionalities to reify social expectations as CArtAgO [32] artifacts, that are made available to the agents in the environment where they are situated. Currently, 2COMM supports social relationship-based agent programming for JADE [4] and JaCaMo [7] agents through two dedicated connectors. 2COMM provides the classes for representing commitments, maintained into artifacts, as well as roles. Moreover, the infrastructure automatically handles the commitment progressions according to the events occurring in the environment. In particular, events that are relevant for the progression of commitments are encoded as *social facts*. 2COMM maintains a trace of all the social facts asserted in a given interaction and updates the involved commitments accordingly.

3.3 Background on Social Commitments

We propose to model interaction explicitly by means of *social commitments* [22, 35], intended as first-class objects that can be used for agent programming. A social commitment $\mathsf{C}(x, y, s, u)$ models the directed relation between two agents: a *debtor* x and a *creditor* y. The debtor commits to its creditor to bring about the consequent condition u when the antecedent condition s holds. Both conditions are conjunctions or disjunctions of events and commitments, and concern the *observable* behavior of the agents, as advocated in [36] for social relationships among autonomous parties.

In this chapter we will assume that antecedent and consequent conditions are expressed in *precedence logic* [37, 38], an event-based linear temporal logic for modeling web services' composition. It deals with occurrences of events along runs (i.e., sequence of instanced events). Event occurrences are assumed to be non-repeating and persistent: once an event has occurred, it has occurred for the whole execution. The logic has three operators: "$\vee$" (choice), "$\wedge$" (co-occurrence), and "$\cdot$" (before). The *before* operator allows constraining the order with which two events must occur; e.g., $a \cdot b$ means that a must occur before b, but the two events do not need to occur one immediately after the other.

Commitments are proactively created by debtors, who, by doing so, manifest both the agent's engagement to bringing about the consequent condition, and the agent's awareness of the condition (antecedent condition) after which it will be held to bring about the consequent. Generally, the antecedent condition will be a condition of interest for the debtor. When the creditor has some degree of control on such a condition, the commitment becomes a tool supporting the interaction. For instance, let us consider the commitment $\mathsf{C}(customer,$

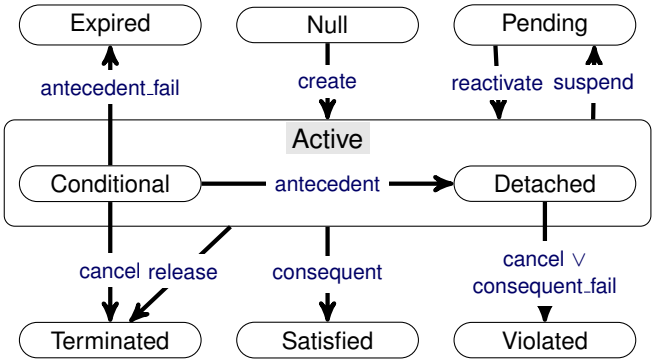


Fig. 3.3: Commitment life cycle [39].

$merchant$, purchaseItemsShipped, purchaseItemsPaid). Here, $merchant$, who is interested in being paid, can leverage the commitment by $customer$ to do so after the shipment: by making purchaseItemsShipped become true, the commitment is detached and, then, $customer$ is held to pay, thus making purchaseItemsPaid become true.

Commitments have a standardized lifecycle, formalized in [39] and depicted in Fig. 3.3. As soon as a commitment is created, it is *active*. Active commitments can further be in two sub-states: *conditional* when neither the antecedent, nor the consequent have occurred, and *detached* as soon as the antecedent becomes true. A commitment is *violated* either when its antecedent is true, but its consequent becomes false or when it is canceled by the debtor when detached. Conversely, it is *satisfied* when the consequent occurs and the engagement is accomplished. It is *expired* when it is no longer in effect; i.e., if the antecedent condition becomes false. Finally, a commitment becomes *terminated* when the creditor releases the debtor from it or when it is canceled, being still conditional.

Social commitments can be manipulated by the agents through some standard operations: namely, *create*, *cancel*, *release*, *discharge*, *assign*, and *delegate* [35]. *Create* instantiates a commitment, changing its status from *null* to *active*. Only the debtor of a commitment can create it. Conversely, *cancel* revokes the commitment, setting its status to *terminated* or *violated*, depending on whether the previous status was *conditional* or *detached*, respectively. Again, only the debtor can cancel a commitment. The rationale is that a commitment debtor is allowed to change its mind about the accomplishment of the consequent only until the antecedent has not occurred, yet. *Release*, in turn, allows the creditor (and only the creditor) to terminate an active commitment. Release does not mean success or

failure of the given commitment, but simply eliminates the expectation put on the debtor. *Discharge* satisfies the commitment. In accordance with [35], we assume that this operation is performed concurrently with the actions that lead to the consequent occurrence. *Delegate* changes the debtor of a given commitment and can be performed only by the new debtor. *Assign*, finally, transfers the commitment to another creditor and can be performed only by the actual one. An active commitment can be further suspended, i.e., put in the *pending* status, by means of the *suspend* operation. Conversely, *reactivate* sets a pending commitment as active, again.

Commitments have a *normative value* because the debtor of a detached commitment is expected to bring about, sooner or later, the consequent condition of that commitment otherwise it will be liable for a violation. The normative value of commitments, i.e. the fact that debtors should satisfy them, creates social expectations on the agents' behavior. A commitment is taken by a debtor towards a creditor on its own initiative. Agents can decide whether satisfying the obligation entailed by the commitment, once detached. An agent creates commitments towards other agents while it is trying to achieve its goals (or precisely with the aim of achieving its goals) [40]. The creation of a commitment starts an interaction of the debtor with its creditor that coordinates, to some extent, the activities of the two, thus supporting the achievement of goals that an agent alone could not achieve. Considering *interaction*, the difference between obligations and commitments, as norms, is that an obligation is a system level norm while a commitment is an agent level norm. At system level, something happens and an obligation is created on some agent. At the agent level, an agent creates a conditional social commitment towards some other agent, based on its own beliefs and goals [40]. The creditor agent will detach the conditional commitment if and when it deems it useful to its own purposes, thus activating the obligation of the debtor agent. So, conditional commitments play a fundamental role in the realization of interactivity, intended as the fact that *a message relates to previous messages and to the way previous messages related to those preceding them* [41]. In other words "there is a causal path from the establishment of a commitment to prior communications by the debtor of that commitment. Obligations by contrast can be designed in or inserted by fiat" [22, Sec. 4.4].

Notably, social commitments are often used for attributing semantics to interaction protocols (e.g. [42–47]). Differently from these works, we use commitments to realize a *relational representation of interaction*, where agents, by their own action, directly create normative binds (represented by social commitments) with one another, and use them to coordinate their activities.

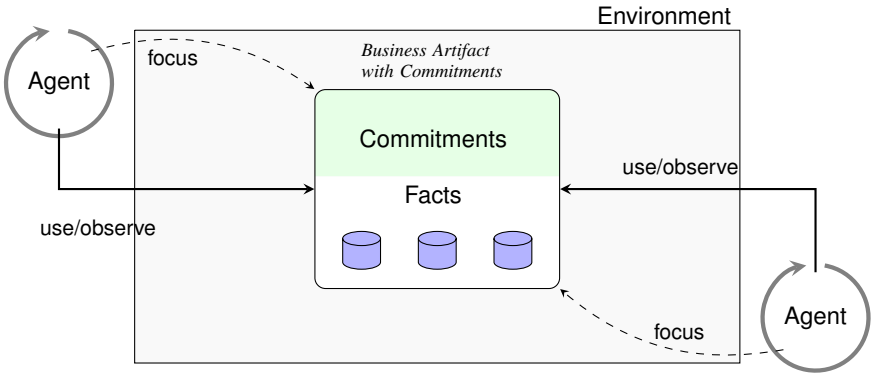


Fig. 3.4: Conceptual view of commitment mediated interaction.

3.4 Business Artifacts with Social Commitments

Figure 3.4 shows a conceptual schema of the proposed solution. Agents interact by using and observing shared artifacts that, besides data (i.e. facts), also encapsulate the commitments created by the agent themselves. More precisely, when an agent needs to interact with others, it is asked to focus on an appropriate artifact available in the environment. Possibly, an agent can be focused on more artifacts at the same time. As a consequence of the focus operation, the agent has access to the state of the artifact, that is, all the information involving the existing commitments and facts that have been asserted, so far, as a consequence of the operation performed upon the artifact itself. Indeed, the focus gives also the agent the capability of acting upon the artifact by creating new commitments or by making other existing commitments progress by asserting facts that corresponds to their antecedent/consequent conditions. By focusing on an artifact, the agent will be notified of any change occurring in the artifact state, such as the assertion of new facts, and the progression of the state of some commitments.

To make this discussion more concrete, let us consider again our simple purchase scenario. We show that when the interaction between the merchant and the customer is expressed in terms commitments, the agents have the means for reasoning upon their engagements, and hence can deliberate the proper operations for making the interaction progress towards the result they are interested to.

First of all, the environment encompasses a *Purchase* artifact, see Fig. 3.5. Both the merchant and the customer need to include such artifact in their scope for having access to it and also for being notified of the changes occurred to it (*focus*). Each agent has its own goal. The merchant aims at gaining some money,

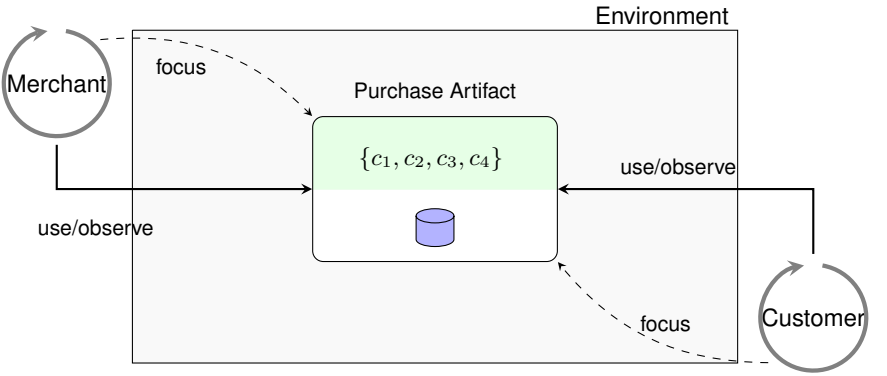


Fig. 3.5: The conceptual view of the purchase scenario.

whereas the customer aims at having some needed items. The *Purchase* artifact represents the medium through which the interaction between the two agents can take form. The expectations about the interaction are shaped as an evolving set of commitments. So, for instance, by committing to pay for the items, should the merchant deliver them, the customer makes clear which expectations the merchant can legitimately draw on its counterpart of the interaction. The created commitment is stored within the *Purchase* artifact. This is exactly the kind of information that is missing in approaches based on (business) artifacts where the normative dimension is not explicitly represented. Agents just react to signals or state changes, but cannot figure out a global picture of the interaction and take advantage of this picture for deciding how to intervene. By relying on commitments as manipulable resources, instead, the agents have the means for creating expectations on how the others will behave as a consequence of their actions, and hence can decide the best course of action that brings them to their goal.

The normative power of commitments emerges not only by the possibility of creating expectations, but also by detecting when these expectations are not met, causing thus a commitment violation. In fact, when the customer commits to pay for the goods, the merchant gains the right to claim against the customer in case the payment is not performed. In other terms, the actions the merchant does upon the *Purchase* artifact may put obligations on the customer, who is pushed to act so as to satisfy her commitments lest incur in violations which may lead to sanctions.

The whole purchase can be effectively modeled by the following set of commitments:

- c_1 : C($merchant, customer, c_2$, purchaseItemsShipped)
- c_2 : C($customer, merchant$, purchaseItemsShipped, purchaseItemsPaid)

- c_3 : C($merchant, customer$, purchaseQuotationRequested, purchaseQuotationProvided)
- c_4 : C($merchant, customer$, purchaseItemsPaid, purchaseReceiptDelivered)

Commitments c_1, c_3 and c_4 are to be created by the merchant at the beginning of the interaction. In particular, c_1 is a *nested* commitment, that is, a commitment that includes other commitments in its conditions. Specifically, c_1 expresses the promise, by the merchant, of delivering the items in case the customer promises to pay for them (commitment c_2). When the customer actually creates c_2, on the one side c_1 is detached, and consequently the merchant is driven to ship the items; on the other hand, the customer promises to the merchant to pay when these items will be delivered. Commitment c_3 is used to model the negotiation phase, it expresses that, should the customer request a quotation for the purchase, the merchant will answer with the quotation. Finally, c_4 is the commitment from the merchant to the customer to send the receipt upon the payment. Note that the events mentioned in the antecedent and consequent of the commitments, are indeed facts (i.e., data) asserted within the artifact state as a consequence of predefined operations made available to the agents by the artifact itself. Therefore, by acting upon the *Purchase* artifact, both merchant and customer have the chance not only to create commitments, but also the make them progress along their lifecyle. In the example, the commitments completely shape the expected evolution of the interaction between the involved agents by making explicit the behavior they are expected to stick to. As such, commitments provide a standard to define patterns of interaction mediated by the environment in which agents are situated.

Objective Coordination and Decoupling. Interaction is surely one of the most critical dimension to be addressed in distributed systems. To cope with the problem, message passing protocols are widely used in many multiagent platforms (see e.g., JaCaMo and JADE), and even in BPMN. A disadvantage of message protocols, however, is that the interaction logic is intermingled with the agent control logic. That is, the coordination is modeled in a *subjective* way. This lack of concern separation hampers software modularity and reuse. For promoting software modularity, coordination should be implemented *on the side of the callee* [48]. Namely, the coordination should be implemented outside the agents, in a class/resource that is accessed concurrently by the agents being coordinated.

Our proposal, on the contrary, enables a form of *objective coordination* [27, 49], where coordination is addressed outside the interacting agents. Objective coordination enables a clear separation between the implementations of the business logic and of the coordination logic by explicitly representing the environment where the agents operate. In our proposal, we meet this property by reifying

commitments, that possess a normative power, inside a dedicated coordination artifact, which is external to the agents. The interaction logic is thus encoded into a single component (i.e., the coordination artifact), and is not distributed and intermingled within the agents' code. A positive consequence is that the implementation of environment resources (i.e., artifacts) and of the agents' processes can be carried out and verified in *isolation*. That is, not only agents are strongly decoupled with each other, as their interaction is mediated by an artifact, but there is also a form of decoupling between agents and artifacts. The interaction logic encoded within an artifact can in fact be changed without the need to change the agents as far as the artifact interface (i.e., the set operations), remains unchanged. For instance, in the purchase scenario, it may be possible to change the interaction logic by adding a timeout on the detachment of commitment c_1. The rationale would be to release the merchant if the customer does not accept within a finite time interval (which is considered an anomalous situation). This, however, is a characterization of the *Purchase* artifact that has no a direct impact on the participants.

Decoupling enables also the development and verification of agents and artifacts in isolation (see below).

Flexibility. Some approaches rely on automata to model the stages through which an interaction progresses. For instance, the *interaction component* [50,51], for the JaCaMo platform, enables both agent-to-agent and agent-to-environment interaction, providing guidelines of how a given organizational goal should be achieved, with a mapping from organizational roles to interaction roles. Guidelines are encoded in an *automaton-like* shape, where states represent protocol steps, and transitions between states are associated with (undirected) obligations: the execution of such steps creates *obligations* on some agents in the system, which can concern actions performed by the agents in the environment, messages that an agent sends to another agent, and events that an agent can perceive (i.e., events emitted from objects in the environment). Business artifacts [18] are another example where the lifecycle of data are represented as automata. By acting upon these artifacts, processes cause automata state changes. The specification of interaction via automata, however, shows a rigidity that prevents agents from taking advantage of the opportunities and of handling exceptions in dynamic and uncertain multiagent environments [52, 53]. Agents are, in fact, confined to the execution sequences provided by the automaton.

On the contrary, commitments do not impose any strict ordering. As far as any detached commitment progresses to satisfaction, the interaction can be considered successful. It must be noticed that a commitment could even be satisfied

before its detachment, this opens the way to deal with "cases" more flexibly and in a way that is paralleling CMMN. Let us suppose, for instance, that we want to capture an emergency situation where some medical equipment is needed urgently. The customer may create commitment c_2 even before knowing the asked quote. That is, there is not a strict ordering between first asking for a quote and then accepting it, as it may happen in an automaton-based model. More interesting, the merchant, knowing the emergency conditions, and trusting the customer, may ship the equipment even before the customer creates commitment c_2. This means for the merchant to bring about the consequent condition of a commitment that has not even been detached. When eventually the customer creates c_2, such a commitment is already detached, and hence the customer pays for the equipment.

Commitments, thus, when used as directly manipulable resources, enable the agents to decide how and when be engaged in an interaction, in ways that are substantially more flexible than approaches based on message passing and automata.

Comparison with Existing Frameworks. We now briefly compare the proposed approach with two of the most established frameworks for multi-agent systems, namely JADE and JaCaMo, already introduced in Sec. 3.2.

A first positive consequence of an explicit, directly manipulable, representation of social commitments is that agents can be programmed by following a uniform schema that is agent-platform independent. This is not, however, the unique advantage. Table 3.1 synthesizes a comparison between JADE and JaCaMo, highlighting aspects that are improved or added by an explicit representation of commitments.

Table 3.1: Comparison among JADE, JaCaMo, and improvements provided by commitments.

	JADE	**JaCaMo**	**Social Commitments**
Programmable interaction means	X	✓	✓
Notification of social relationships of interaction	X	X	✓
Interaction/agent logic decoupling	X	X	✓
Reasoning on expected behaviors	X	X	✓
Definition of structured schemas for interaction	✓	X	✓
Runtime interaction monitoring	X	✓	✓
Agent programming aware of social relationships	X	X	✓

- *Programmable interaction means*: The reification of social commitments as artifacts embodied in the environment significantly improves JADE, where programmable interaction means are missing. On the other hand,

JaCaMo already integrates the CArtAgO component and, hence, already includes programmable artifacts used as means of interaction.

- *Notification of social relationships of interaction*: Our proposed business artifacts implement the commitment lifecycle. Any change of state in the artifact, thus, corresponds to a change in the interaction state. For this reason, a form of automatic notification to the agents of such changes (concerning social relationships) is enabled. This is not provided by JADE nor by JaCaMo.
- *Interaction/agent logic decoupling*: In JADE and JaCaMo, no specific abstractions are used for modeling and implementing the interaction logic. In JADE, agents interact by exchanging messages. In JaCaMo, agents can modify an artifact for communicating with others. In both cases, however, the semantics of the message or of the artifact operation must be encoded within the interacting agents. With dedicated artifacts encapsulating commitments, when an agent acts upon the environment, the meaning, and implications, of such an action is modeled via a state change occurring in a specific commitment. It follows that the logic of the interaction is not spread inside the agent code, but within the artifacts.
- *Reasoning on expected behaviors*: As we have said, commitments have a normative power. Neither JADE nor JaCaMo can rely upon an analogous mechanism.
- *Definition of structured schemas for interaction*: The commitment lifecycle allows relying on a general schema for programming agents that is independent of the agent-platform.
- *Runtime interaction monitoring*: Although we will not discuss this aspect in detail, the use of dedicated artifacts facilitates the implementation of monitors that supervise the whole interaction occurring in the system. It would be sufficient, in fact, to have an agent monitoring each artifact of and hence capturing every change of state in the commitments. Of course, this is a direct consequence of having social relationships explicitly modeled as first-class elements and not implicitly modeled via message passing as in JADE. In JaCaMo, a monitor could also be implemented as an observer of the artifacts in the environment.
- *Agent programming aware of social relationships*: The proposed approach guides a programmer in developing an agent that is aware and compliant with its engagements, as we will show in detail in the following section.

3.5 Patterns for Programming Agents with Social Relationships

Commitments bring along advantages also from the point of view of agent software development. In fact, their standardized lifecycle can be used as a reference for developing agents that use them. Intuitively, an agent involved in a commitment, either as debtor or as creditor, should act so as to make the commitment state progress toward satisfaction w.r.t. its lifecycle (consequent condition achievement). Thus, the lifecycle can provide a hint to the programmer on what agent behaviors she needs to implement. We can think of the commitment lifecycle as a sort of bank of programming patterns, that can be defined taking also into account the goals the two commitment participants aim at. The relation between goals and commitments has been deeply studied in [40, 54], where the authors propose several rules of practical reasoning that agents can use to determine, e.g., when it is convenient to create a commitment, or detach an existing one, when they should act to satisfy a commitment, and so on. Intuitively, these practical rules help agents to align their intentions by means of commitments, so as that all the agents participating to an interaction can achieve their desired goal by exploiting the cooperation of the others.

We take advantage of these practical rules, and propose some agent programming patterns where, by means of the direct use of commitments as interaction resources, an agent can induce the cooperation from others

The aim of the following patterns is, thus, to implement one (or more) social behaviors that allows an agent ag to achieve a goal G by cooperating with other agents. In the following, we assume that an agent has already focused on an artifact of interest, and use the term *social state* for denoting the state of such an artifact, that is, the collection of commitments and facts (data) maintained within the artifact.

(1) *Entice*:

- *Intent*: Finding the cooperation of another agent by making an offer.
- *Applicability*: When, by inspecting the social state, the agent does not discover cooperation offers by other agents that support the achievement of its goal adequately, then the agent creates an offer itself.
- *Implementation*: Implement a behavior that creates a commitment $\mathsf{C}(ag, other, G, q)$, where ag is the help-seeking agent, $other$ is another agent focused on the artifact, and q is a condition that ag can bring about, either directly or via the cooperation with others.

(2) *Cooperate*:

- *Intent*: taking part to a collaboration by making a commitment progress towards satisfaction.
- *Applicability*: when the agent is either the debtor or the creditor of an active commitment.
- *Implementation*:
  - When ag is the creditor: implement a behavior that is triggered by the creation of a commitment $c : \mathsf{C}(other, ag, p, G)$ and that brings about p, so as to detach c.
  - When ag is the debtor: implement a behavior that is triggered by the detachment of a commitment $c : \mathsf{C}(ag, other, G, q)$ and that brings about q so as to satisfy c.

(3) *Withdraw offer*

- *Intent*: Retire an offer of cooperation that is no longer needed. Agent ag has tried to entice the cooperation of some other agent by creating $\mathsf{C}(ag, other, G, q)$, but now it realizes that G is no longer needed. Retiring the offer, ag avoids to consume resources for achieving q in exchange of the undesired G.
- *Applicability*: When goal G is no longer necessary (local decision making of ag), and $\mathsf{C}(ag, other, G, q)$ is still conditional in the social state.
- *Implementation*: Implement a behavior that cancels the conditional commitment $\mathsf{C}(ag, other, G, q)$ currently stored in the social state.

(4) *Give up*

- *Intent*: Release the obligation upon the achievement of a goal, when such a goal is no longer desired. Agent ag has taken part to cooperation by detaching commitment $c : \mathsf{C}(other, ag, p, G)$. The original intent of ag was to get G, and by performing p it has caused the detachment of c. Now the obligation of bringing about G is pending on $other$. While c is detached, however, ag may realize that G is no longer needed. It may even been deleterious, and so can decide to release the commitment, that is, can remove the obligation on $other$ to bring about G. More generally, a creditor can release a commitment even when the commitment is still conditional: ag is not interested in the offer made by the debtor and brings the commitment to a terminal state.
- *Applicability*: When goal G is no longer necessary, and the active (i.e., either conditional or detached) commitment $c : \mathsf{C}(other, ag, p, G)$ is stored in the social state.
- *Implementation*: Implement a behavior that releases commitment c.

It's worth noting that the applicability of the *Withdraw offer* and *Give up* patterns is the same. The two patterns represents different ways in which debtor and creditor can leverage an existing commitment. Being the commitment still conditional, should the debtor be not interested anymore in the antecedent, it can cancel it by means of *Withdraw Offer*. Conversely, should the creditor be not interested in the consequent, it can release the debtor from its commitment by means of *Give up*.

The above patterns relate agent goals with the engagements the agent is involved in. In particular, *Entice*, *Withdraw offer* and *Give up*, are patterns "from goal to commitment", that is, on the necessity of achieving or dropping a goal, an agent acts directly on a commitment by means the operations create, cancel, and release, respectively. Pattern *Cooperate* encompasses both a stance "from goal to commitment" when an agent is creditor and acts upon a commitment to bring about the antecedent condition, so as to detach it. And it has also a "commitment to goal" stance when the agent is a debtor and react to the detachment of a commitment by adding the consequent condition among the goals it has to pursue. Of course, the patterns we have introduced cover just a subset of the possible state transitions in a commitment lifecycles. Other patterns leading to the violation of a commitment or to its suspension are possible, but for the purpose of the present paper we have focused on those patterns that promote coordination and lead to the satisfaction of a commitment.

Let us explain how these patterns are applied to the previously introduced *Purchase* scenario. For the sake of simplicity, we use an abstract language based on ECA rules (Event, Condition, Action), for expressing the agents' behaviors. In Sec. 3.6 we will show an actual implementation in JaCaMo+ [19] (i.e. JaCaMo extended with 2COMM [10, 34]). Here, we adopt the following syntax for ECA rules: $ON\ event\ IF\ condition\ THEN\ action$. The event denotes the trigger for activating such a rule, in general it is a goal that the agent wants to achieve or an event that must be properly treated by the agent. The condition is a contextual circumstance that must hold for the rule to be actually fired. Finally, the action is a course of operations that modify the environment so as to obtain a desired effect.

Let us take the $merchant$'s perspective, whose ultimate goal is to conclude the purchase and get paid. It can do so by exploiting the *Entice* schema, in such a way to create commitments c_1, c_3 and c_4, defined in Sec. 3.4, to make an offer to the potential customer. This behavior could be captured by the following ECA rule:

ON needMoney IF haveItemsToSell
DO create(c_1 : C($merchant, customer, c_2$, purchaseItemsShipped));
create(c_3 : C($merchant, customer$, purchaseQuotationRequested, purchaseQuotationProvided));
create(c_4 : C($merchant, customer$, purchaseItemsPaid, purchaseReceiptDelivered)).

The use of the *Entice* pattern is strictly connected to the use of the *Cooperate* one (as debtor), to tackle the relevant commitment state changes (i.e. their detachment). For c_1, the corresponding ECA rule would be:

ON detached(c_1) DO $...ship...$ assert(purchaseItemsShipped).

The triggering event is the detachment of commitment c_1, that is, $customer$ has confirmed the purchase by creating c_2. As a reaction, the agent asserts the social fact denoting that the items have been shipped. That is, the effect of the behavior would be the satisfaction of commitment c_1. By applying the *Cooperate* pattern, the $customer$ can be provided with those behaviors needed to handle the detachment of c_3 and c_4, as well.

The $merchant$ could also want to withdraw the items from the sale, thereby closing the purchase, for instance if the $customer$ does not react within three days (because it's a limited-time offer). In this case, the commitments will still be in the state Conditional, a fact that we denote through conditional(.). We can model such a behavior by means of the *Withdraw offer* pattern, as follows:

ON threeDaysPassed IF conditional(c_1) $\wedge$ conditional(c_3) $\wedge$ conditional(c_4);
DO cancel(c_1 : C($merchant, customer, c_2$, purchaseItemsShipped));
cancel(c_3 : C($merchant, customer$, purchaseQuotationRequested, purchaseQuotationProvided));
cancel(c_4 : C($merchant, customer$, purchaseItemsPaid, purchaseReceiptDelivered)).

In this case the commitments made by the $merchant$ are no longer needed, and can be canceled.

Let us now take the $customer$'s perspective, which has several possibilities. For instance, being interested in buying a large amount of items, upon creation of c_3, the agent could apply the *Cooperate* pattern (as creditor), and request a quotation.

ON create(c_3) IF interestedInLargeAmount DO assert(purchaseQuotationRequested).

Otherwise, if it is not interested in receiving a quotation (maybe because it's a regular buyer who knows the price in advance), it could simply confirm the purchase, both applying the *Entice* and *Cooperate* patterns w.r.t. the creation of c_2 and subsequent detachment of c_1.

ON create(c_1) IF interestedInItems DO create(c_2).

Also in this case, upon creation of c_2, the $customer$ should be equipped with the behavior(s) needed to handle its detachment.

ON detached(c_2) IF true DO ...$send$ EPO... assert(purchaseItemsPaid).

The satisfaction of c_2 by $customer$, finally allows the $merchant$ to satisfy its original goal. Conversely, should $customer$ be not interested in the $merchant$'s offer, it could decide to release the latter from its engagement coming from c_1, by applying the *Give up* pattern.

ON create(c_1) IF not interestedInItems DO release(c_1).

3.6 The Hiring Process Scenario

We exemplify the proposal in a scenario which is well-known in the field of business processes.

Example 3.2 (Hiring Process [55]). A hirer opens a call for a job position for which many candidates will likely apply over a time period. As long as the position remains open, each candidate is called for an interview, and then evaluated. The evaluation of a single candidate takes time, even weeks. Two processes are involved, namely Hiring Process and Evaluate Candidate. They are synchronized by means of a data store representing the job status in a database. Hiring Process updates the data store when the job is opened, filled or abandoned. Evaluate Candidate queries it immediately upon instantiation. Hiring Process just waits for a message from Evaluate Candidate, indicating that an applicant has been selected for the position, and has accepted the offer. Evaluate Candidate has many instances, each of which carries out the evaluation of a single candidate. Once the job is filled new applicants just receive a position closed message, and any still running instances of evaluate candidate are terminated.

The scenario is an instance of the *one-to-many* pattern of coordination between process instances [24, 55], occurring when the multiplicity of an activity is not in accordance with the multiplicity of another one. In this case, it is necessary to separate them into different interacting processes.

Figure 3.6 shows the solution proposed by Silver, consisting of a *Hiring Process*, each of whose instances tackles a single job opening, an *Evaluate Candidate process*, whose instances tackle each a different candidate, and an *Applicant Process*, whose instances amount to candidates. While the relationship between the instances of *Evaluate Candidate* and *Applicant* is *one-to-one*, the relationship between the instances of the *Hiring Process* and those of the *Evaluate Candidate* process is inherently *one-to-many*. The states of all these processes are to be coordinated and, in particular, the *Hiring Process* must have a way to enable *Evaluate Candidate* when a new job is opened, and to disable its running instances when

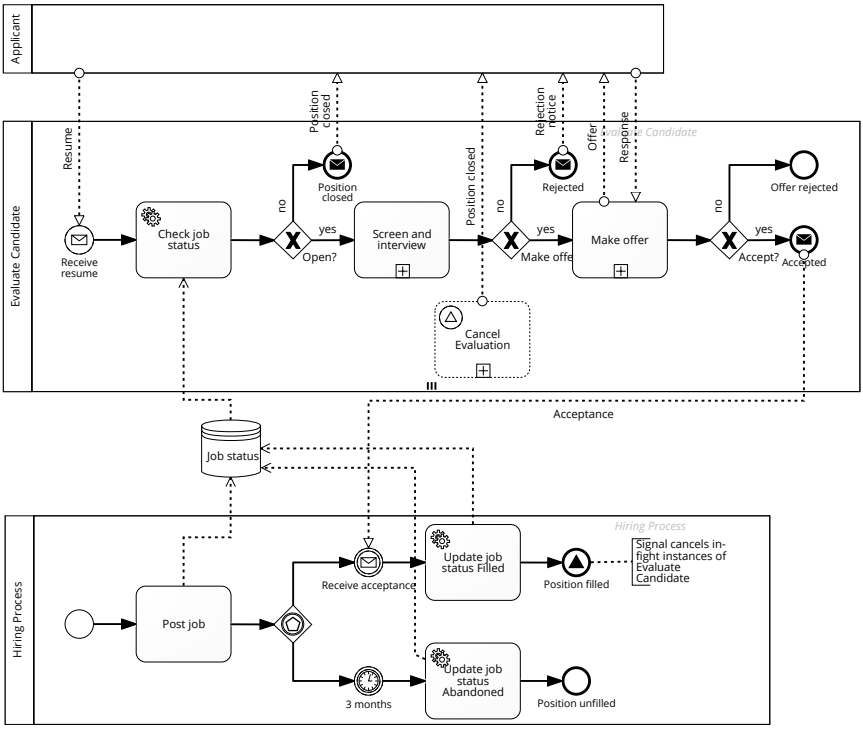


Fig. 3.6: The *Hiring Process* scenario represented in BPMN [55].

the job position has been assigned. To this end, a data storage is introduced, which is external to the processes, but to which all of them have access, supporting synchronization and data consistency. Silver's solution highlights the limitations of BPMN in modeling the coordination, already discussed in Sec. 3.2. In fact, with reference to Fig. 3.6, the BPMN representation only suggests that the *Hiring Process* should update the status of a job opening after receiving the acceptance of an offer by some candidate, and that this will let *Evaluate Candidate* know that it accomplished its task for that opening.

We now explain how to use social commitments to realize the Hiring Process. In Fig. 3.7, hi (hirer), ev_i (evaluator), and i (candidate) represent the agents of this scenario, that observe and use artifacts in a shared environment. For each available position, there will be just one agent playing the hirer role, whereas many evaluators and candidates will be possible. For simplicity, we assume that each candidate i will be evaluated by a specific evaluator ev_i; this does not

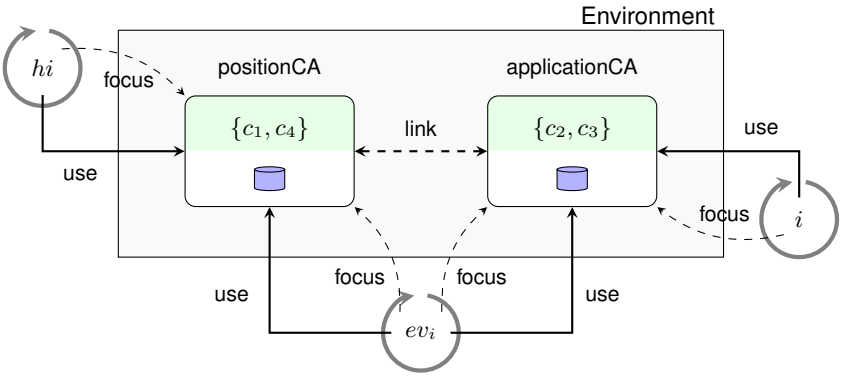


Fig. 3.7: The *Hiring Process* scenario. Commitments $c_1 - c_4$ refer to the set in Fig. 3.8.

exclude, however, that the same agent be an evaluator for different candidates. The workspace includes also two business artifacts extended with social commitments. Each of them stores both the social commitments and the facts that are relevant for the application. The artifact positionCA, in particular, maintains the state of the position, it is instantiated just once, and it is only accessed by the hirer and the evaluators. Instead, the artifact applicationCA is instantiated once for each candidate, each instance keeping track of the status of the application made by that specific candidate.

The Business Artifacts extended with Social Commitments. The normative layer of the two artifacts is expressed by the commitments listed in Fig. 3.8. In particular, c_1 and c_4 will characterize positionCA, while c_2 and c_3 will be maintained inside applicationCA. By operating on the two artifacts agents can make the interaction evolve following the lifecycle of the four commitments. For instance, hi can open the position by creating c_4, thereby enticing an interaction with ev_i. The two artifacts are linked, so the operations performed by the agents make the commitments maintained in both artifacts progress. The meanings of the commitment are as follows.

- *Commitment* c_1. It encodes that ev_i is committed to carry out the evaluation for the application by candidate i according to a predefined procedure. The procedure, outlined in evaluate-candidate$_{ev_i}$, is equivalent to the *Evaluate Candidate* process in Fig. 3.6. The sequence position-filled$_{hi}$ · msg-position-closed$_{ev_i}$, describes that the evaluator informs candidate i that the position is closed as soon as the position is assigned by hi. On the BPMN

c_1 : $\mathsf{C}(ev_i, hi, \text{post-job}_{hi} \cdot \text{apply}_i, \text{post-job}_{hi} \cdot \text{apply}_i \cdot \text{evaluate-candidate}_{ev_i})$

c_2 : $\mathsf{C}(ev_i, i, \text{post-job}_{hi} \cdot \text{apply}_i, \text{post-job}_{hi} \cdot \text{apply}_i \cdot \text{inform-outcome}_{ev_i})$

c_3 : $\mathsf{C}(i, ev_i, \text{make-offer}_{ev_i}, \text{make-offer}_{ev_i} \cdot (\text{response-yes}_i \vee \text{response-no}_i))$

c_4 : $\mathsf{C}(hi, ev_i, \text{post-job}_{hi} \cdot (\text{accepted}_{ev_i} \vee \text{timeout\_3months}_{hi}), \text{post-job}_{hi} \cdot \text{hiring}_{hi})$

Where:

$$\text{evaluate-candidate}_{ev_i} \equiv \text{position-filled}_{hi} \cdot \text{msg-position-closed}_{ev_i} \vee (\text{screen-interview}_{ev_i} \cdot (\text{msg-rejection-notice}_{ev_i} \vee \text{make-offer}_{ev_i} \cdot (\text{response-yes}_i \cdot \text{accepted}_{ev_i} \vee \text{response-no}_i \cdot \text{offer-rejected}_{ev_i})))$$

$$\text{inform-outcome}_{ev_i} \equiv (\text{msg-position-closed}_{ev_i} \vee \text{msg-rejection-notice}_{ev_i} \vee \text{make-offer}_{ev_i})$$

$$\text{hiring}_{hi} \equiv (\text{accepted}_{ev_i} \cdot \text{position-filled}_{hi}) \vee (\text{timeout\_3months}_{hi} \cdot \text{position-abandoned}_{hi})$$

Fig. 3.8: The set of commitments included in the normative layer for the *Hiring Process* scenario.

process, this is equivalent to the *Check Job Status* activity and the consequent message sending in case the position has already been assigned, and it also models the capturing of signal *position filled* sent by the hirer while this evaluator is still processing an application. The rest of the condition, $(\text{screen-interview}_{ev_i} \ldots \text{offer-rejected}_{ev_i})$ encodes all the possible branches in the execution of process *Evaluate Candidate*, including the messages sent to and received from other roles. It is important to note the shape of the antecedent and of the consequent conditions of c_1, among which a temporal relation is captured. Indeed, ev_i's commitment is detached (and hence the agent will have to bring about the evaluation process) only when a job position has been posted, and a candidate has applied for it. In order to model that ev_i is expected to evaluate candidate i only after this has applied for the position, the antecedent condition (i.e., $\text{post-job}_{hi} \cdot \text{apply}_i$) occurs as a prefix in the consequent condition. A similar pattern is used also in the following commitments.

- *Commitment* c_2. Evaluator ev_i takes also commitment c_2 towards candidate i to take into account the application and to provide i with an answer for the application. Such a commitment has the same antecedent condition of c_1. The answer can either be a message informing that the position has already been closed, a rejection notice, or even an offer for the job. Also in this case, apply_i is used in order to make $\text{inform-outcome}_{ev_i}$ follow the satisfaction of the antecedent condition.

- *Commitment* c_3. This commitment is pretty interesting from our point of view. It represents candidate i's promise to answer either "yes" or "no" to an eventual offer made by ev_i. The BPMN in Fig. 3.6 does not specify the internal behavior of the candidate, so an answer for the offer cannot be taken for granted. Indeed, the candidate may never answer; the evaluator would not be able to detect this anomalous situation, and may, thus, await indefinitely. In our opinion, this example highlights the weaknesses of BPMN in modeling the coordination of independent processes. Certainly, one could enrich the evaluator's process so as to wait a predefined time interval for an answer. But this is just a way for handling the exception. With commitments, instead, our major concern is to stimulate the agents to act so as to make the interaction progress. ev_i, for instance, could create c_2 only after the creation of c_3. In this way, the evaluator could legitimately expect an answer, should an offer be made. When candidate i is offered the job, it is stimulated to answer either "yes" or "no" due to the existence of commitment c_3. Any anomalous situations in which the candidate does not answer is clearly detected by the violation of c_3. The evaluator process does not need to capture this eventuality directly.
- *Commitment* c_4. Finally, commitment c_4 represents the engagement of the hirer towards the evaluator, and in particular describes the process carried out by the hirer in Fig. 3.6. The antecedent condition expresses the start event of the process (i.e., $\mathsf{post\text{-}job}_{ev_i}$) followed by two alternative events that enables the hirer to complete the process. Such a commitment is to be created by the hirer to entice the cooperation of the evaluator as soon as a job is posted. The first event is $\mathsf{accepted}_{ev_i}$, meaning that an evaluator has found a suitable candidate. The second event is $\mathsf{timeout\_3months}_{hi}$, which stands for the complementary event to $\mathsf{position\text{-}filled}_{ev_i}$. Therefore, after a period of three months, when $\mathsf{timeout\_3months}_{hi}$ occurs, the position cannot be assigned anymore. The consequent condition of c_4 describes how the hirer completes the process depending on what event has satisfied the antecedent condition. In case of event $\mathsf{accepted}_{ev_i}$, hi assign the position and notify this to all the evaluators possibly still running ($\mathsf{position\text{-}filled}_{ev_i}$). Notably, this allows the evaluators still active to bring about event $\mathsf{msg\text{-}position\text{-}closed}_{ev_i}$ so as to discharge both commitments c_1 and c_2. In case the antecedent has been satisfied by event $\mathsf{timeout\_3months}_{hi}$, instead, the position status is updated to abandoned.

As an illustration, we now show how it is possible to develop agents amounting to hirer, evaluators, and candidates leveraging the programming patterns presented

in Sec. 3.5. To this end we rely on the 2COMM platform, already introduced in Sec. 3.2, and we discuss a realization of the agents written in JaCaMo[1].

Programming the Agents. JaCaMo extended with 2COMM [10,34] allows implementing agents through the Jason language, as sets of plans expressed as ECA-like rules. In particular, each agent has a *belief base*, a set of ground (first-order) atomic formulas which represent the state of the world according to the agent's vision, and a *plan library*. Moreover, it is possible to specify *achievement* (operator '!') and *test* (operator '?') goals. A Jason plan is specified as $triggering\_event : context \leftarrow body$, where the *triggering_event* denotes the event the plan handles (which can be either the addition or the deletion of some belief or goal), the *context* specifies the circumstances when the plan could is applicable, and the *body* is the course of action that should be taken. With 2COMM, the triggering event can be a state change occurring in some commitment.

We realize the three processes hi, ev_i, and i as Jason agents. Listing 3.1 reports the implementation fo hi. The first plan, at Line 1, means that when hi has the goal of posting a job position (and eventually finding a suitable candidate), it *entices* the cooperation of ev_i by creating commitment c_4. Indeed, condition $\mathsf{accepted}_{ev_i}$, the hirer would like to become true, cannot be met by hi on its own. It needs the cooperation of some evaluator, to take care of the applications. The evaluator will create commitment c_1, and the hirer (being c_1's creditor) will then be in condition to apply the first case of the *cooperate* pattern, by contributing to making the antecedent condition of c_1 become true. This is done by posting a job (Line 3). For detaching the commitment it is necessary that some candidate applies for the position. The last plan, at Line 6, is instead realized by using the second case of the *cooperate* pattern: the hirer, being c_4's debtor, has to properly react as soon as the commitment gets *detached*. So, hi updates the position status depending on the events that have occurred (acceptance or timeout), satisfying the antecedent. It's worth noting that the first and the last plan are strictly linked. The latter allows the agent to satisfy the commitment created in the former, once detached. The absence of the second would make the agent unable to fulfill its engagement, and, thus, liable in case of violation.

The code of the evaluator (see in Listing 3.2) is a bit more complex since it encompasses the whole evaluation process. The first plan, at Line 1, follows both the *cooperate* (as creditor) and *entice* patterns. By reacting to the creation c_4, the agent creates c_1 establishing its availability to perform an evaluation. The second plan, in turn, is triggered when the candidate entices an interaction with the

[1] A running implementation of the *Hiring Process* in JaCaMo, together with the source code of 2COMM and other examples can be found at http://di.unito.it/2comm.

```
+!post-job <- createC4[artifact_id(positionCA)].

+cc(ev_i, hi, post-job · apply, post-job · apply · evaluate-candidate, CONDITIONAL)
  <- postJob[artifact_id(positionCA)].

+cc(hi, ev_i, post-job · (accepted ∨ timeout3months), post-job · hiring, DETACHED)
   : Result
  <- updatePosition(Result)[artifact_id(positionCA)].
```

Listing 3.1: Hirer hi.

```
+cc(hi, ev_i, post-job · (accepted ∨ timeout3months), post-job · hiring, CONDITIONAL)
  <- createC1[artifact_id(positionCA)]

+cc(i, ev_i, make-offer, make-offer · (response-yes ∨ response-yes), CONDITIONAL)
  <- createC2[artifact_id(applicationCA)]

+cc(ev_i, hi, post-job · apply , apply · evaluate-candidate, DETACHED)
  <- screenInterview[artifact_id(applicationCA)];
     // ... Choice; !offerOrReject(Choice).
+!offerOrReject(Choice) : Choice == yes
  <- makeOffer[artifact_id(applicationCA)].
+!offerOrReject(Choice) : Choice == no
  <- rejectionNotice[artifact_id(applicationCA)]; releaseC2.

+responseYes : cc(ev_i, hi, post-job · apply, apply · evaluate-candidate, DETACHED)
  <- offerAccepted[artifact_id(positionCA)]
+responseNo : cc(ev_i, hi, post-job · apply, apply · evaluate-candidate, DETACHED)
  <- offerRejected[artifact_id(positionCA)].

+cc(ev_i, hi, post-job · apply , apply · evaluate-candidate, DETACHED)
   : position-filled | position-abandoned
  <- positionClosed[artifact_id(applicationCA)]; releaseC3.

+position-filled
   : cc(ev_i, i, post-job · apply, post-job · apply · inform-outcome, CONDITIONAL)
  <- cancelC2; releaseC3.
+position-abandoned
   : cc(ev_i, i, post-job · apply, post-job · apply · inform-outcome, CONDITIONAL)
  <- cancelC2; releaseC3.
```

Listing 3.2: Evaluator ev_i.

```
+!find-job <- createC3.

+post-job <- apply[artifact_id(applicationCA)].

+cc(candidate_i, ev_i, make-offer, response-yes ∨ response-no, DETACHED)
  <- // ... Choice; !response(Choice).
+!response(Choice) : Choice == yes
  <- responseYes[artifact_id(applicationCA)].
+!response(Choice) : Choice == no
  <- responseNo[artifact_id(applicationCA)].
```

Listing 3.3: Candidate i.

evaluator by creating c_3. In this case the evaluator reacts creating c_2, so as to encourage the candidate to send an application. The third plan, in which the evaluator reacts to the detachment of commitment c_1 by interviewing the candidate, provides the agent with the ability to handle the evolution of c_1, likely bringing it to satisfaction. To this end, the agent acts upon applicationCA by executing screenInterview. After this operation, the evaluator comes up with a Choice, either accept the candidate or reject it. This choice will, thus, activate a proper behavior (see the plans at Lines 10–13). It is worth noting that, in case of rejection, the agent can also apply the *give up* pattern, by releasing candidate i from its engagement due to c_3. Since an offer won't be made, a response is no longer needed. Plans at Lines 15–18 are, instead, used to react to a candidate's answer, either "yes" or "no", to a possible offer. Accordingly, the evaluator performs an operation on positionCA so as to complete the evaluation process and, then, discharge its commitment c_1. The plan starting at Line 20 captures the situation in which the evaluator is held to inform the candidate as soon as the position gets filled. Interestingly, no specific rule is requested for treating commitment c_2 because whenever such a commitment gets detached, the evaluator satisfies it by satisfying c_1. However, at a normative layer, commitment c_2 is fundamental to detect the misbehavior of the evaluator towards the candidate. Should the position be filled or abandoned with c_2 still conditional, the evaluator can also be equipped with those plans to cancel its commitment no longer needed, following the *withdraw offer* pattern (see Lines 24 and 27).

Finally, Listing 3.3 sketches the pseudocode of a candidate. As soon as the agent has a goal of finding a job, it can create c_3, so as to entice the cooperation of the evaluator. Then, when a job is posted candidate i can decide to send an application (see the plan at Line 3). The last three plans are needed to react to the detachment of commitment c_3 by answering either "yes" or "no" to an offer.

3.7 Conclusions

In this chapter we have addressed one of the challenges in engineering distributed systems: the development of high-level abstractions for modeling interaction. The distribution of goals over a number of interacting and independent execution threads, in fact, demands for special interaction abstractions that allow complex systems to scale up effectively.

In this chapter, we explain the adoption of commitments, which enjoy several interesting properties, for explicitly modeling interaction at a higher level than message passing protocols. Commitments have a normative power, and hence enable agents to create expectations about the behaviors of others. They can be

created and manipulated directly by the agents via a standard set of operations, and this allows agents to reason on how to act upon commitments in order to coordinate with others. Recent proposals, see [56], exploit commitments to enable the design of socio-technical systems, composed of both social (people and organizations) and technical (computers and networks) elements, that satisfy the stakeholders' requirements, and their refinement through design patterns.

We have seen that commitments can be reified in interaction resources, and that they can be manipulated by the agents in autonomy. From a software engineering point of view, there are several advantages. First of all, the modularity is substantially improved since there is a clear separation between the interaction logic, encapsulated within a proper artifact, and the agent logic. This, in turn, promotes software reuse. More importantly, the standard lifecycle of commitments can be the base for developing a number of agent programming patterns. The rationale is that an agent involved in a commitment should possess at least those behaviors for making that commitment evolve toward satisfaction.

Also in the context of industrial applications, multiagent systems proved to be effective in enabling adaptability and flexibility of automated production systems (see, e.g., [57]). Agent platforms are nowadays widely used in industrial domains, ranging from smart environments to logistics. Some of them were developed in industrial settings, as is the case of JADE, which was developed at TIM TILab, the research center of the main Italian telecommunication company. The proposed commitment-based approach, being implemented as a library that can easily be adapted to multiple agent-frameworks, paves the way for a fruitful exploitation of the advantages coming from an explicit representation of social relationships in several application domains.

Although property verification falls outside the scope of this chapter, there are a number of solutions that can be exploited thanks to the use of commitments. Indeed, as a consequence of the induced decoupling, one can verify the correctness of both agents and artifacts in isolation. On the artifact side, it is possible to verify general properties on the commitments that may be possibly created by the agents. For instance, [58] presents the 2CL methodology, an approach for specifying business protocols in a declarative way with social commitments. Being a formal language, 2CL enables several forms of verification. For instance, it is possible to analyze the space of all possible evolutions of a set of commitments in order to determine risky situations. That is, conditions where commitments are likely to be violated. In [42], the authors propose a methodology for checking, at design time, whether a commitment protocol satisfies a predefined set of generic properties such as consistency, effectiveness and robustness. A commitment algebra is presented in [59], where it provides a conceptual basis for reasoning about

protocols in terms of notions like refinement or aggregation, and includes the operators merge and choice and a subsumption relation for protocols.

On the agent side, similar automatic verification techniques can be used to assess the compliance of an agent to a given commitment protocol; see for instance [60]. Moreover, commitments can used as the basis of a dynamic, agent type-checking system which, at execution time, can assess whether an agent is equipped with a set of behaviors that is appropriate for carrying out an interaction [61].

The approach presented in this chapter sets the basis for an important further development: the scaling from commitment to accountability. Accountability has a very broad range of interpretations. Briefly, we can think of accountability as a directed relationship from an account-giver to an account-taker, that results from an agreement between the parties. When an accountability relationships is established, the account-taker can legitimately ask, under some agreed conditions, an account about a process of interest to the account-giver. On the other hand, the account-giver is legitimately required to provide the account to the account-taker. There are several differences between social commitments and accountability. First of all, a social commitment results from the initiative of its debtor and does not require the agreement of its creditor.[2] In addition, social commitments imply *liability*, i.e., when a social commitment is violated the creditor is legitimated to complain against the debtor [63]. However, the creditor has no right to ask for an explanation about the violation. This hampers the transmission of information that the creditor could use to deal with the unexpected situation brought about by the violation. We deem that a computational model of accountability [64–67] is the key to develop feedback/reporting frameworks, similarly to what is often done in human organizations, that can be exploited for handling both anomalous conditions and new opportunities. In [68] an early conceptual view on how accountability can be the means to reach robustness is discussed. Specifically, the idea is that accountability relationships can be used, at design time, to model feedback channels through which information can be conveyed from the agent who produces it, to the agent who can exploit such information in its internal deliberative cycle. This general principle is at the basis of robustness in distributed systems, since it provides the information they need to properly handle a perturbation.

[2] The notion of commitments introduced by [62] does require acceptance from its creditor, but this is not the interpretation we have used in this paper which is aligned with the view presented in [35].

References

[1] M. J. Wooldridge, *Introduction to multiagent systems*. Wiley (2002), ISBN 978-0-471-49691-5.

[2] M. N. Huhns and L. M. Stephens, *Multiagent Systems and Societies of Agents*, chap. 2. MIT Press, ISBN 0262232030, pp. 79–120 (1999), ISBN 0262232030.

[3] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr and A. Ricci, A survey of programming languages and platforms for multi-agent systems, *Informatica* **30**, 1, pp. 33–44 (2006).

[4] F. L. Bellifemine, F. Bergenti, G. Caire and A. Poggi, JADE - A Java Agent Development Framework, in *Multi-Agent Programming: Languages, Platforms and Applications*, *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Vol. 15. Springer, pp. 125–147 (2005).

[5] A. Omicini and F. Zambonelli, TuCSoN: a coordination model for mobile information agents, in *Proc. of IIIS'98*. IDI – NTNU, Trondheim (Norway), pp. 177–187 (1998).

[6] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings and J. Treur, Desire: Modelling Multi-Agent Systems in a Compositional Formal Framework, *Int. J. of Cooperative Information Systems* **06**, 01, pp. 67–94 (1997).

[7] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci and A. Santi, Multi-agent oriented programming with JaCaMo, *Science of Computer Programming* **78**, 6, pp. 747–761 (2013).

[8] I. J. Timm, T. Scholz, O. Herzog, K. Krempels and O. Spaniol, From agents to multiagent systems, in *Multiagent Engineering, Theory and Applications in Enterprises*. Springer, pp. 35–51 (2006).

[9] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Z. Kwiatkowska, J. A. McDermid and R. F. Paige, Large-scale complex IT systems, *Communications of the ACM* **55**, 7, pp. 71–77 (2012), doi:10.1145/2209249.2209268, http://doi.acm.org/10.1145/2209249.2209268.

[10] M. Baldoni, C. Baroglio and F. Capuzzimati, A Commitment-based Infrastructure for Programming Socio-Technical Systems, *ACM Transactions on Internet Technology* **14**, 4, pp. 23:1–23:23 (2014).

[11] M. P. Singh, A social semantics for agent communication languages, in *Issues in Agent Communication*, *Lecture Notes in Computer Science*, Vol. 1916. Springer, pp. 31–45 (2000).

[12] C. Castelfranchi, *Principles of Individual Social Action*. Kluwer, pp. 163–192 (1997).

[13] A. K. Chopra, A. Artikis, J. Bentahar, M. Colombetti, F. Dignum, N. Fornara, A. J. I. Jones, M. P. Singh and P. Yolum, Research directions in agent communication, *ACM Transactions on Intelligent Systems Technology* **4**, 2, pp. 20:1–20:23 (2013), doi:10.1145/2438653.2438655, https://doi.org/10.1145/2438653.2438655.

[14] Y. Demazeau, From interactions to collective behaviour in agent-based systems, in *Proceedings of the 1st. European Conference on Cognitive Science*. Saint-Malo, pp. 117–132 (1995).

[15] D. Weyns, A. Omicini and J. Odell, Environment as a first class abstraction in multiagent systems, *JAAMAS* **14**, 1, pp. 5–30 (2007).

[16] A. Omicini, A. Ricci and M. Viroli, Artifacts in the A&A meta-model for multi-agent systems, *Autonomous Agents and Multi-Agent Systems* **17**, 3, pp. 432–456 (2008).
[17] A. Ricci, M. Piunti and M. Viroli, Environment programming in multi-agent systems: an artifact-based perspective, *Autonomous Agents and Multi-Agent Systems* **23**, 2, pp. 158–192 (2011).
[18] A. Nigam and N. S. Caswell, Business artifacts: An approach to operational specification, *IBM Systems Journal* **42**, 3, pp. 428–445 (2003).
[19] M. Baldoni, C. Baroglio, F. Capuzzimati and R. Micalizio, Commitment-based Agent Interaction in JaCaMo+, *Fundamenta Informaticae* **159**, 1–2, pp. 1–33 (2018).
[20] M. A. Sirbu, Credits and debits on the internet, *IEEE Spectrum* **34**, 2, pp. 23–29 (1997).
[21] Object Management Group (OMG), Business Process Model and Notation (BPMN), Version 2.0, OMG Document Number formal/2011-01-03 (https://www.omg.org/spec/BPMN/2.0/PDF) (2011).
[22] M. P. Singh, Commitments in multiagent systems some controversies, some prospects, in *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, chap. 31. College Publications, London, pp. 601–626 (2011).
[23] Object Management Group (OMG), Case Management Model and Notation (CMMN), Version 1.1, OMG Document Number formal/2016-12-01 (https://www.omg.org/spec/CMMN/1.1/PDF) (2016).
[24] M. Dumas, On the convergence of data and process engineering, in *Proc. of Advances in Databases and Information Systems*, *LNCS*, Vol. 6909. Springer, pp. 19–26 (2011).
[25] M. Baldoni, C. Baroglio, F. Capuzzimati and R. Micalizio, Objective Coordination with Business Artifacts and Social Engagements, in *Business Process Management Workshops*, *Lecture Notes in Business Information Processing*, Vol. 308. Springer, pp. 71–88 (2018).
[26] K. Bhattacharya, R. Hull and J. Su, *A data-centric design methodology for business processes*, Handbook of Research on Business Process Modeling. IGI Publishing, pp. 503–531 (2009).
[27] A. Omicini and S. Ossowski, Objective versus subjective coordination in the engineering of agent systems, in *AgentLink*, *LNCS*, Vol. 2586. Springer, pp. 179–202 (2003).
[28] F. Zambonelli, N. R. Jennings and M. Wooldridge, Developing multiagent systems: The Gaia methodology, *ACM Trans. Softw. Eng. Methodol.* **12**, 3, pp. 317–370 (2003).
[29] M. Wooldridge, N. R. Jennings and D. Kinny, The GAIA methodology for agent-oriented analysis and design, *Autonomous Agents and multi-agent systems* **3**, 3, pp. 285–312 (2000).
[30] M. de Brito, J. F. Hübner and O. Boissier, A conceptual model for situated artificial institutions, in *Computational Logic in Multi-Agent Systems*. Springer, pp. 35–51 (2014).
[31] R. H. Bordini, J. F. Hübner and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons (2007), ISBN 0470029005.
[32] A. Ricci, M. Piunti, M. Viroli and A. Omicini, *Environment Programming in CArtAgO*. Springer US, Boston, MA, ISBN 978-0-387-89299-3, pp. 259–288 (2009), ISBN 978-0-387-89299-3.

[33] J. F. Hubner, J. S. Sichman and O. Boissier, Developing organised multiagent systems using the MOISE+ model: Programming issues at the system and agent levels, *Int. J. Agent-Oriented Softw. Eng.* **1**, 3/4, pp. 370–395 (2007).

[34] M. Baldoni, C. Baroglio, R. Micalizio and S. Tedeschi, Programming agents by their social relationships: A commitment-based approach, *Algorithms* **12**, 4, p. 76 (2019).

[35] M. P. Singh, An ontology for commitments in multiagent systems, *Artif. Intell. Law* **7**, 1, pp. 97–113 (1999).

[36] M. Dastani, D. Grossi, J. C. Meyer and N. A. M. Tinnemeier, Normative multi-agent programs and their logics, in *Normative Multi-Agent Systems, 15.03.–20.03.2009*, *Dagstuhl Seminar Proceedings*, Vol. 09121. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2009).

[37] M. P. Singh, Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition, in *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14–18, 2003, Melbourne, Victoria, Australia, Proceedings*. ACM, pp. 907–914 (2003).

[38] E. Marengo, M. Baldoni, C. Baroglio, A. Chopra, V. Patti and M. Singh, Commitments with regulations: reasoning about safety and control in REGULA, in *Proc. of the 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, Vol. 2, pp. 467–474 (2011).

[39] P. R. Telang and M. P. Singh, Specifying and Verifying Cross-Organizational Business Models: An Agent-Oriented Approach, *IEEE Trans. Services Computing* **5**, 3, pp. 305–318 (2012).

[40] P. R. Telang, M. P. Singh and N. Yorke-Smith, Relating Goal and Commitment Semantics, in *Post-proc. of ProMAS*, *LNCS*, Vol. 7217. Springer (2011).

[41] S. Rafaeli, *Sage Annual Review of Communication Research: Advancing Communication Science: Merging Mass and Interpersonal Processes*, chap. (Chapter 4) Interactivity: From new media to communication. Sage (1988).

[42] P. Yolum and M. P. Singh, Commitment Machines, in *Intelligent Agents VIII, 8th Int. WS, ATAL 2001*, *LNCS*, Vol. 2333. Springer, pp. 235–247 (2002).

[43] N. Fornara and M. Colombetti, Defining Interaction Protocols using a Commitment-based Agent Communication Language, in *Proc. of the Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003)*. ACM, pp. 520–527 (2003).

[44] A. K. Chopra, *Commitment Alignment: Semantics, Patterns, and Decision Procedures for Distributed Computing*, Ph.D. thesis, North Carolina State University, Raleigh, NC (2009).

[45] P. Torroni, F. Chesani, P. Mello and M. Montali, Social Commitments in Time: Satisfied or Compensated, in *Declarative Agent Languages and Technologies VII, 7th International Workshop (DALT 2009)*, Vol. 5948, pp. 228–243 (2010).

[46] M. El-Menshawy, J. Bentahar and R. Dssouli, Verifiable semantic model for agent interactions using social commitments, in *Languages, Methodologies, and Development Tools for Multi-Agent Systems, Second International Workshop, LADS 2009, Torino, Italy, September 7–9, 2009, Revised Selected Papers*, no. 6039 in Lecture Notes in Computer Science. Springer, pp. 128–152 (2009).

[47] M. Baldoni, C. Baroglio, E. Marengo and V. Patti, Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach, *ACM Trans. on*

Intelligent Sys. and Tech., Special Issue on Agent Communication **4**, 2, pp. 22:1–22:25 (2013).

[48] M. Philippsen, A survey of concurrent object-oriented languages, *Concurrency - Practice and Experience* **12**, 10, pp. 917–980 (2000).

[49] M. Schumacher, *Objective Coordination in Multi-agent System Engineering: Design and Implementation*. Springer-Verlag, Berlin, Heidelberg (2001), ISBN 3-540-41982-9.

[50] M. R. Zatelli, A. Ricci and J. F. Hübner, Integrating interaction with agents, environment, and organisation in JaCaMo, *IJAOSE* **5**, 2/3, pp. 266–302 (2016).

[51] O. Boissier, R. H. Bordini, J. F. Hübner and A. Ricci, Dimensions in programming multi-agent systems, *The Knowledge Engineering Review* **34**, p. e2 (2019).

[52] P. Yolum and M. P. Singh, Flexible protocol specification and execution: applying event calculus planning using commitments, in *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15–19, 2002, Bologna, Italy, Proceedings*. ACM, pp. 527–534 (2002).

[53] M. Winikoff, W. Liu and J. Harland, Enhancing commitment machines, in *Declarative Agent Languages and Technologies II, Second International Workshop, DALT 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, no. 3476 in Lecture Notes in Computer Science. Springer, pp. 198–220 (2004).

[54] P. Telang, M. Singh and N. Yorke-Smith, A coupled operational semantics for goals and commitments, *Journal of Artificial Intelligence Research* **65**, pp. 31–85 (2019).

[55] B. Silver, *BPMN Method and Style, with BPMN Implementer's Guide*, 2nd edn. Cody-Cassidy Press, Aptos, CA, USA (2012).

[56] Ö. Kafali, N. Ajmeri and M. P. Singh, DESEN: specification of sociotechnical systems via patterns of regulation and control, *ACM Trans. Softw. Eng. Methodol.* **29**, 1, pp. 7:1–7:50 (2020).

[57] VDI, VDI/VDE 2653 Blatt 1 - Multi-agent systems in industrial automation - Fundamentals, Tech. Rep., VDI - The Association of German Engineers - Department of Industrial Information Technology (2018).

[58] M. Baldoni, C. Baroglio, E. Marengo, V. Patti and F. Capuzzimati, Engineering commitment-based business protocols with the 2CL methodology, *Autonomous Agents and Multi-Agent Systems* **28**, 4, pp. 519–557 (2014).

[59] A. U. Mallya and M. P. Singh, An algebra for commitment protocols, *Auton. Agents Multi Agent Syst.* **14**, 2, pp. 143–163 (2007).

[60] D. D'Aprile, L. Giordano, A. Martelli, G. L. Pozzato, D. Rognone and D. Theseider Duprè, *Information Systems: Crossroads for organization, management, accounting and engineering*, chap. Business process compliance verification: An annotation based approach with commitments, pp. 563–570 (2012).

[61] M. Baldoni, C. Baroglio, F. Capuzzimati and R. Micalizio, Type Checking for Protocol Role Enactments via Commitments, *Journal of Autonomous Agents and Multi-Agent Systems* **32**, 3, pp. 349–386 (2018).

[62] C. Castelfranchi, Commitments: From Individual Intentions to Groups and Organizations, in *Proceedings of the First International Conference on Multiagent Systems (ICMAS)*. The MIT Press, San Francisco, California, USA, pp. 41–48 (1995).

[63] A. K. Chopra and M. P. Singh, From social machines to social protocols: Software engineering foundations for sociotechnical systems, in *Proc. of the 25th Int. Conf. on WWW* (2016).

[64] M. Baldoni, C. Baroglio, K. M. May, R. Micalizio and S. Tedeschi, Computational Accountability, in *Deep Understanding and Reasoning: A challenge for Next-generation Intelligent Agents, URANIA 2016*, Vol. 1802. CEUR, Workshop Proceedings, pp. 56–62 (2016).

[65] M. Baldoni, C. Baroglio, O. Boissier, K. M. May, R. Micalizio and S. Tedeschi, Accountability and Responsibility in Agents Organizations, in *PRIMA 2018: Principles and Practice of Multi-Agent Systems*, no. 11224 in Lecture Notes in Computer Science. Springer, pp. 261–278 (2018).

[66] M. Baldoni, C. Baroglio, K. M. May, R. Micalizio and S. Tedeschi, MOCA: An ORM MOdel for Computational Accountability, *Journal of Intelligenza Artificiale* **13**, 1, pp. 5–20 (2019a).

[67] M. Baldoni, C. Baroglio, O. Boissier, R. Micalizio and S. Tedeschi, Engineering Business Processes through Accountability and Agents, in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13–17, 2019*. International Foundation for Autonomous Agents and Multiagent Systems, pp. 1796–1798 (2019b).

[68] M. Baldoni, C. Baroglio and R. Micalizio, Fragility and Robustness in Multiagent Systems, in *To appear in the 8th International Workshop on Engineering Multi-Agent Systems (EMAS 2020) @ AAMAS 2020, 8–9 May 2020, Auckland, New Zealand* (2020).

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0004

Chapter 4

Intelligent Agents are More Complex: Initial Empirical Findings

Gal A. Kaminka* and Alon T. Zanbar†
Bar-Ilan University, Israel
**galk@cs.biu.ac.il*, †*atzanbar@gmail.com*

4.1 Introduction

For many years, significant research efforts have been spent on investigating methodologies, tools, models and technologies for engineering autonomous agents software. Research into agent architectures and their structure, programming languages specialized for building agents, formal models and their implementation, development methodologies, middleware software, have been discussed in the literature, encompassing multiple communities of researchers, with at least partial overlaps in interests and approaches.

The fundamental assumption underlying these research efforts is that such specialization is *needed*, because autonomous agent software poses engineering requirements that may not be easily met by more general (and more familiar) software engineering and programming paradigms. Specialized tools, models, programming languages, code architectures and abstractions make sense, if the software engineering problem is specialized.

A broad overview of the literature reveals that for the most part, the truth of this assumption has been supported by qualitative arguments and anecdotal evidence. Agent-oriented programming [1] is by now a familiar and accepted programming paradigm, and countless discussions of its merits and its distinctiveness with respect to other programming paradigms (e.g., object-oriented programming, aspect-oriented programming) are commonly found on the internet. Agent architectures are commercially available as development platforms and are incorporated into products. Indeed, agent-oriented software development methodologies are taught and utilized in and out of academic circles [2–5].

However, there is a disturbing lack of quantitative, empirical evidence for the distinctiveness of autonomous agent software. Lacking such evidence, agent software engineers rely on intuition, experience, and philosophical arguments when they evaluate or advocate specialized methods.

This paper provides *the first empirical evidence for the distinctiveness of autonomous agent software*, compared to other software categories. We quantitatively analyze over 500 software projects: 140 autonomous agent and robotics projects (from RoboCup, the Agent Negotiations Competitions, Chess, and other sources), together with close to 400 automatically selected software projects from github, of various types. With each, we utilize general source code metrics, such as *Cyclomatic Complexity*, *Cohesion*, *Coupling*, and others used by general software engineering researchers to quantify meaningful characteristics of software (over 250 measures, see below).

We conducted both statistical and machine-learning analysis, to determine (1) whether agents emerge as a distinguishable sub-group within the pool, and (2) whether there are clear distinguishing measures. We find that agent software is *clearly* and *significantly* different from other types of software of comparable size. This result appears both when using manual statistical analysis, as well as machine learning methods. Specifically, autonomous agents software is significantly more complex (in the sense of control flow complexity) than other software categories. We discuss potential implications of these results.

4.2 Background

There is vast literature reporting on software engineering of autonomous agents: agent architectures, agent-oriented programming languages, formal models and their implementation, development methodologies, middleware software, and more. We cannot do justice to these efforts for lack of space. For brevity, we use the term *agent-oriented software engineering* (AOSE) to refer to the combined research area, encompassing the collective efforts of the various communities engaged in relevant research. We emphasize no bias in the selection of this name, and With due apologies to all the different threads of work whose unique contributions are blurred by our choice.

AOSE is a thriving area of research, with at least one dedicated annual conference/workshop and a specialized journal[1] [1, 2, 6–9]. For the most part, the arguments for the study of AOSE as distinct from general software

[1]International Journal of Agent-Oriented Software Engineering.

engineering are well argued *philosophically*, and *qualitatively* pointing out inherent conceptual differences between the software engineering of agents. To the best of our knowledge, little quantitative empirical evidence — certainly not at the scale detailed below — has been offered to support these important conceptual arguments.

Closely related, pioneering works into software engineering in robotics (e.g., [10–16] similarly argue *qualitatively* for distinguishing software engineering in robotics. Some emphasize specific middleware frameworks (e.g., [10–12, 17–19]), while others focus on critical capabilities or approaches [20–23]). These are detailed and well-reasoned arguments, however the underlying implicit assumption is similar to those in AOSE: that robotics software is sufficiently different from general software, that it merits its distinct methodologies and tools to ease software development. Indeed, we report below that robot code is similar in some aspects to autonomous agents code, but is not as easily distinguished from general software.

The rarity of quantitative investigations in AOSE (see below for notable exceptions) is not for lack of quantitative methods in *general* software engineering. Beginning with the 1970s pioneering research on Cyclomatic Complexity [24] and Halstead measures [25] there have been many investigations both proposing quantitative metrics of software constructs, and relating the measurements to software quality, development effort, software type, and other attributes of interest [26–28]. For example, metrics such as *Cyclomatic Complexity*, *Coupling*, and *Cohesion* — generated from analysis of the software source code and the program control flow graph — have been shown to correlate with defects [24, 29, 30]. Maintaining their values within specific ranges (or below some thresholds) tends to lower the expected *defect creation rate*, and improve other measures of software quality. Development and exploration of software metrics continues today, e.g., for paradigms such as aspect-oriented programming [31]. See [32] for a comprehensive survey.

Software metrics have been used to classify software, or cluster together software based on measured characteristics, as we do in this paper [33]. For example, De Souza and Maia defined software metric thresholds based on context ([34]). [35] showed this approach is applicable for Android projects. Another example can be found in [36], who found linkage between the size and complexity of open source projects, to attractiveness of the project for contributors.

Surprisingly, despite the prevalence and usefulness of software metrics as noted above, the use of software metrics in intelligent agent and robot

software remains limited, and is not generally reported in relevant literature. Perhaps this is due to lack of data, or the relative novelty of products, which leads to sparse and relative rare expertise in commercial-grade development. AOSE-specific software metrics, specialized to agent programming paradigms and languages, have been proposed in AOSE research circles [37–40], often specific to agent-oriented programming languages (e.g., 2/3APL, JASON). Because of their specialized use cases, which prohibit their use in general software, we were reluctant to use them in this study, which uses general metrics to contrast software from many different categories.

4.3 Software Project Data Collection and Curation

We begin with an overview of the data collection and curation process. The data collected will be used in the analysis processes described in Secs. 4.4–4.5.

4.3.1 *Data Sources*

RoboCup. RoboCup is one of the oldest and largest annual global robotics competition events in the world — taking place since 1997. The event is organized in several different divisions. Within each division, there are multiple leagues, with their own rules. For example, within the soccer division, there were over the years up to three different simulation-based leagues (*2D*, *3D*, and *coach*), and several physical robot competitions (standard platform, small-size, mid-size, and two humanoid leagues). The competitions themselves are between completely autonomous agents/robots; no human in the loop. In most cases, the agents run in completely distributed fashion, without a centralized controller.

The bulk of the code in the various leagues is written by graduate students and researchers in robotics and artificial intelligence, some from top universities in these fields. The simulation leagues follow an internal rule, which requires all teams to release a binary version of their code within a year following the competition. Source code release is not required, but strongly encouraged. Indeed, we use the source code from many 2D simulation league teams, downloaded from their repository server. In addition, we used source code from other RoboCup soccer leagues, gathered from the internet.

Automated Negotiating Agent Competition (ANAC). The annual International Automated Negotiating Agents Competition (ANAC) is used by the automated negotiation research community to benchmark and evaluate its work and to challenge itself. The benchmark problems and evaluation results and the protocols and strategies developed are available to the wider research community. ANAC has similar properties to the RoboCup in the sense of emphasizing autonomous agents. It is a popular competition for software agent researchers, maintains a requirement that all the sources of the agents participating in the competition are made available for research. We collected ANAC software agent projects from the competition web site.

Additional data. We additionally found open source robotics projects from the DARPA Grand and Urban challenges, and from industrial projects where our lab was involved in research.

4.3.2 *Automatic Data Harvesting*

From the sources above, we first collected agent and robot software projects — all we could find and use: 2D RoboCup teams for which source code is available, the ANAC agent projects, robotics software from RoboCup and the other sources described above. We extended the search for relevant software to github projects tagged *chess*, as problem solving is a close AI domain.

GitHub has more than 24 million users and more than 67 million code repositories. It is the largest repository of open source projects in the world. GitHub exposes robust API for finding repositories using extensive query language, which we used to find relevant project for analysis. Repositories in GitHub are categorized by users using tags, which we used to categorize software projects.

The process of collecting and filtering of repositories from GitHub was *automatic*, as described in Fig. 4.1. The primary constraint in selecting software projects is comparability. The source code collected for agents uses C, C++, and Java, and so we restricted ourselves to projects in these languages, to prevent language-specific bias in the metrics. Similarly, we restricted ourselves to software size (measured in lines of code — LOC) in comparable ranges.

- Programming languages : C, C++, Java

- high Level of maturity
- Distinct classification in github (for github projects)
- Size > 900 lines of code (LOC)

As a control group to the software projects above (focusing on AI and agents), we similarly harvested software projects in domains very different from agents or AI. Table 4.1 shows a breakdown of the number and categories of the harvested software projects in the dataset (almost a terabyte). In total, there were 118 projects generally classified as autonomous agents for software or virtual environments, 20 projects classified as autonomous robots, and 377 projects in other categories. Table 4.2 lists the minimum, maximum and median project size in each domain, measured in LOC.

Table 4.1 Software project data breakdown.

Classification	Source	Software Domain	Size	Maturity Indicator
Autonomous Agents	RoboCup 2D simulation	Virtual Robots	64	Qualification for RoboCup
	ANAC	Negotiating Agents	26	Qualification for ANAC
	GitHub	Chess-Playing Engines	28	> 5 GitHub stars
General	GitHub	Audio	54	> 5 GitHub stars
		Education	50	
		Finance	26	
		Games	34	
		Graphics	60	
		IDE	53	
		Mobile Applications	42	
		Security	58	
Robots	DARPA Challenges	Autonomous Car	2	Qualification for Challenge
	RoboCup competition	Soccer Physical Robots	15	
	Applied R&D Projects	Robots	3	

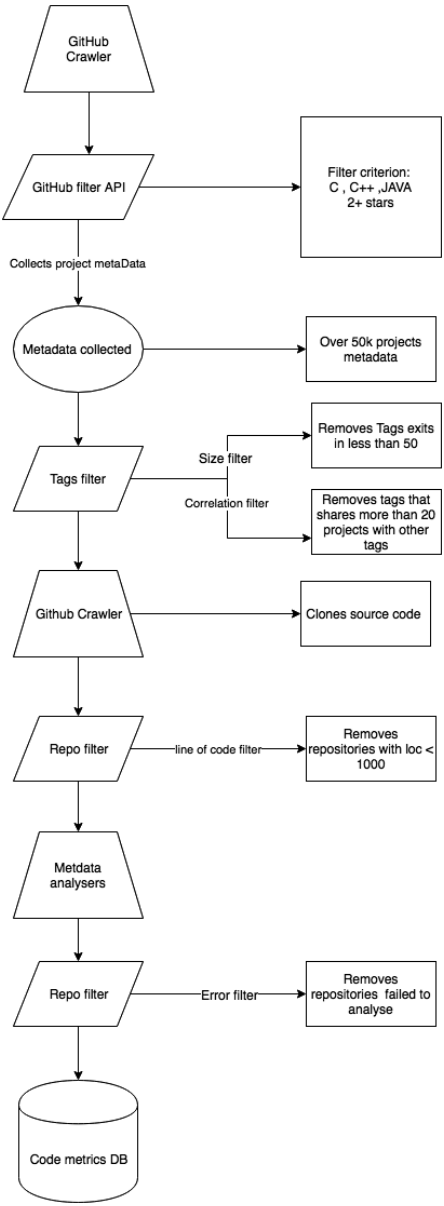


Fig. 4.1 Automatic flow of selecting GitHub projects.

Table 4.2 Software projects min, max, and median LOC.

Software Domain	Min, Max, Median Size in LOC
Virtual Robots	1010, 153661, 23495
Negotiating Agents	1031, 102816, 1352
Chess-Playing Engines	1084, 59108, 5311
Audio	1065, 1912860, 17010
Education	1026, 393360, 6933
Finance	1136, 450524, 10455
34	1393, 185784, 5064
Graphics	1168, 385036, 18769
IDE	1457, 401897, 32486
Mobile Applications	1210, 129366, 4658
Security	1214, 164228, 10341
Autonomous Car	117848, 117848, 117848
Soccer Physical Robots	15335, 793966, 54895
Robots	3131, 64028, 10588

4.3.3 *The Measurement Pipeline*

The essence of the process is the measurement, i.e., the generation of measurements from applying code metrics to the software. We focus on source code metrics in this paper. The source code of each project was processed to extract two different data structures: a control flow graph, and a code statistics database. These, in turn, are used to calculate several different metrics. Additionally we save information on the context of the repository (name, location, category) and other information like source code language, competition results, the year in which the code was deployed, etc.

We used two different tools, independently, to allow validation of the results: CCCC[2] and Analizo.[3] The two tools were run on two 24-core XEON servers, each with 76GB of ram. Total CPU time is more than a month.

The measurement tools provide the following general software metrics, for different level of analysis (see [32] for detailed descriptions). As with the restriction on choice of language, we are restricted to using general metrics as they allow for measuring non-agent code. Otherwise, we'd be able to use code metrics specific to AOSE [37–40], and specialized languages (e.g., 2/3APL, JASON).

[2]http://cccc.sourceforge.net/.
[3]http://www.analizo.org/.

Summary & Project Level Metrics:

- Total Lines of Code (total_loc)
- Total Number of Modules (total_modules)
- Total Number of Methods (total_nom)

Module Level Metrics:

- Afferent Connections per Class (ACC)
- Average Cyclomatic Complexity per Method (ACCM)
- Average Method Lines of Code (AMLOC)
- Average Number of Parameters (ANPM)
- Coupling Between Objects (CBO)
- Coupling Factor (COF)
- Depth of Inheritance Tree (DIT)
- Lack of Cohesion of Methods (LCOM4)
- Lines of Code (LOC)
- Number of Attributes (NOA)
- Number of Children (NOC)
- Number of Methods (NOM)
- Number of Public Attributes (NPA)
- Number of Public Methods (NPM)
- Response for Class (RFC)
- Structural Complexity (SC)

We collected not only the raw metrics above, but also their aggregation in various ways, so as to minimize the inherent loss of information. Thus for each metric, we also computed its mean, mode, minimum value, maximum value, quantiles (lower, max, median, min, ninety five, upper), standard deviation, variance, skewness, and kurtosis. All in all, each software project was represented by more than 250 measurements.

4.4 Statistical Analysis

We conducted two separate analysis efforts which had common general goal. This section details the results of a statistical analysis, while the next section presents the use of machine-learning analysis. The focus in both is to reveal differences, if they occur, between the different software categories, as expressed in the measurements of different metrics.

Every project is represented by approximately 250 different metrics. As such, it is difficult to attempt to find differentiating metric by hand. We therefore used a heuristic procedure to assist in finding promising features. Algorithm 1 describes the procedure. We emphasize that this is a heuristic procedure, to draw human attention to features of interest, not for statistical inference.

The idea is to iterate over the software domains. For each domain r, we separate it out from the others, and then use a two-tailed t-test to contrast the distribution of the metric values in the domain and in all others. For example, one iteration of the algorithm would run two-sample t-test between the values of accm_mean of projects in the *RoboCup 2D* against the values of accm_mean of all projects in the control group (tagged as 'non agent').

A lower p value from the t-test is used as a heuristic, indicating that potentially a good differentiating feature has been detected. We collect all the domains differentiated by the metric f into a common set indexed by f. We then look for sets larger than two. We use a threshold to avoid distractions from a metric that may distinguish a specific domain from all others, by chance.

Algorithm 1 Common differentiator algorithm

for all $r \in Domains$ **do**
  $others \leftarrow (Domains - \{r\})$
  for all $f \in metrics$ **do**
    if *2-tailed t-test*$(r_f, others_f) < 0.05$ **then**
      $CommonSet_f \leftarrow CommonSet_f \bigcup r$
for all $f \in metrics$ **do**
  if $|CommonSet_f| >= 3$ **then** ▷ 3 or more clustered together?
    $selected_f \leftarrow CommonSet_f$
return all $selected_f$

Table 4.3 shows the output of the algorithm for each individual metric, when listed in increasing order of probability (i.e., in order of *decreasing* indication of separation power). The top four metrics are the ACCM mean and its upper and median quantiles, and the Coupling Factor (CoF) metric, which measures coupling between modules. These four metrics clearly distinguish between the agent domains (RoboCup 2D Simulation, Chess, ANAC agents).

Table 4.3 The top distinguishing features in descending order, and the software domains they cluster together.

Metric	Repositories	p Value
accm_mean	[RoboC-2D, Chess, ANAC]	1.18E-04
accm_quantile_upper	[RoboC-2D, Chess, ANAC]	8.78E-04
accm_quantile_median	[RoboC-2D, Chess, ANAC]	1.17E-03
total_cof	[RoboC-2D, Chess, ANAC]	1.19E-03
noa_skewness	[RoboC-2D, IDE, Graphics]	2.59E-03
nom_quantile_upper	[RoboC-2D, ANAC, Audio]	6.27E-03
amloc_quantile_upper	[RoboC-2D, Chess, ANAC, ...]	7.99E-03
nom_mean	[RoboC-2D, ANAC, Graphics, Audio]	1.07E-02
anpm_quantile_upper	[RoboC-2D, ANAC, Graphics]	1.14E-02
noa_kurtosis	[RoboC-2D, Ide, Games, Graphics]	1.28E-02

We use the p value in Table 4.3 as a heuristic indicator for the human analyst. It gives an indication of the strength of the clustering, independent of the content of the cluster. Even if the agent domains could be distinguished from the others, we could easily expect other software domains to be so clustered. However, the fact is that the strongest distinguishing metrics put autonomous agents together, apart from other domains.

We then moved to examining the results visually, using box-plots to display the distribution of specific metrics of each software domain. We seek features which, as clearly as possible, distinguish the three classes of domains.

Indeed some metrics clearly are different between domains. For example, Fig. 4.2 show the box-plot distribution of the Lack-of-Cohesion (LCOM4) metric, which received generally low rank by the heuristic procedure (i.e., a relative high p value). Here, we clearly see that the *RoboCup-Other-Leagues* group stands out, compared to the other software domains. However, it is the only domain in the cluster, and does not distinguish the agents or robots domains from others.

Other metrics may sometimes cluster together more than one domain, but are not able to distinguish agents from non-agents code. For example, Fig. 4.3 show the distribution of the Structural Complexity metric. We can see that the inner-quantile range and median are similar between *RoboCup 2D* and *Other RoboCup Leagues*, suggesting some commonality in behavior of the structure of classes and objects. However, it reveals no commonality between the different domains of the same class (Agents, Robots, or General Software). In other words, it distinguishes some domains from others (to an extent) but does not cluster together domains that come from the same software class.

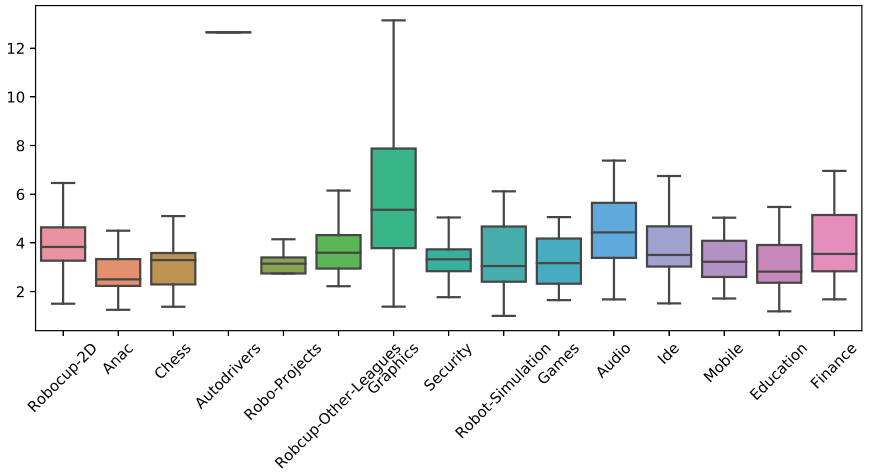


Fig. 4.2 Box plot distribution of LCOM4 means. The vertical axis range is 0–12.

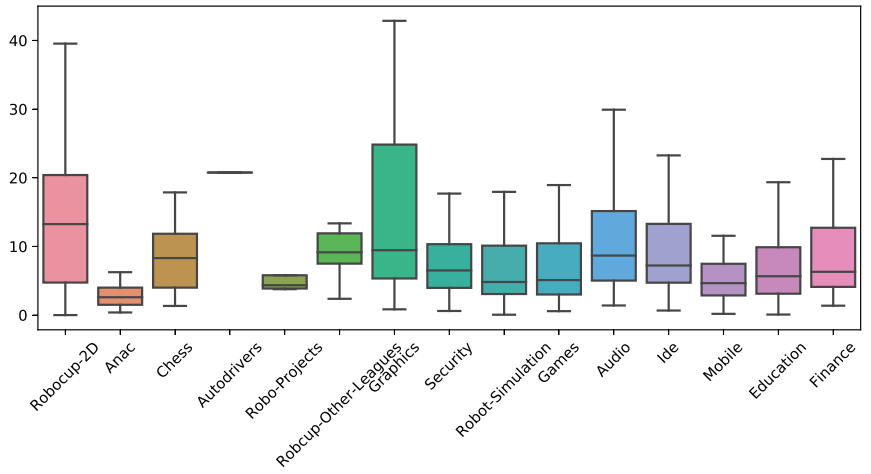


Fig. 4.3 Box plot distribution of the Structural Complexity metric for software domains. The vertical axis range is 0–40. *RoboCup 2D simulation* and *RoboCup-Other-Leagues* have larger variance and higher values than all other software domains.

In contrast, metrics that were ranked high by Alg. 1 visually show much more promise. For example, Table 4.3 suggests the mean ACCM is and the MLOC upper quartile are promising, in terms of their ability to distinguish between agents and non-agent software. Figures 4.4 and 4.5 show the box

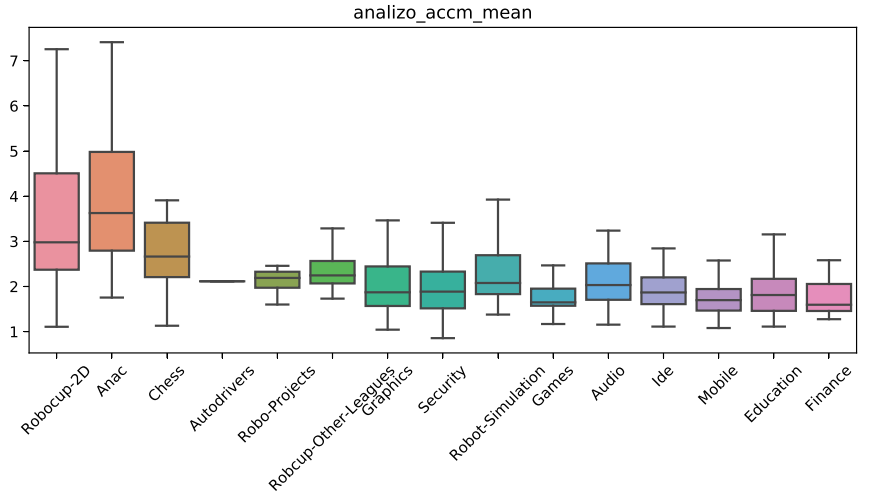


Fig. 4.4 Box plot distribution of mean ACCM of software domains. The vertical axis range is 0–8.

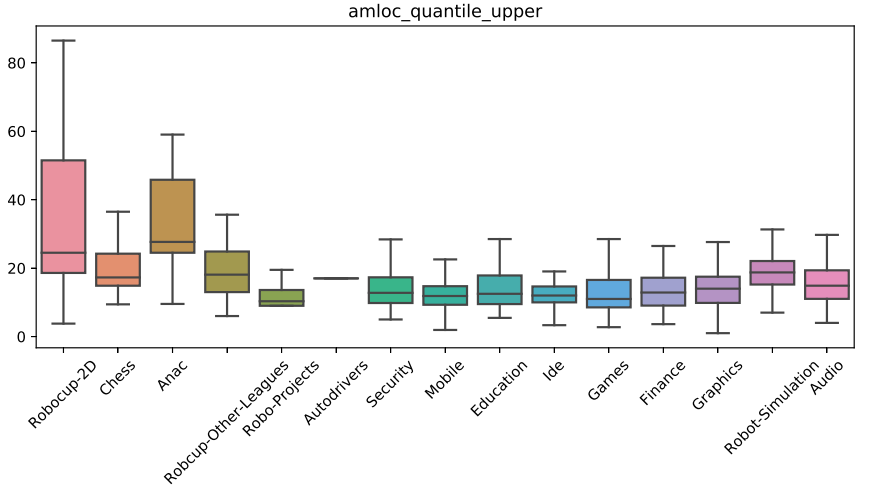


Fig. 4.5 Box plot distribution of AMLOC upper quantile. The vertical axis range is 0–40.

plot distributions these metrics. Visually, in both figures, the box-plots for the Agents class (RoboCup 2D, ANAC, Chess) are clearly prominent relative to other software domains.

Interim Summary. We defer a discussion of the *meaning* of these findings to Sec. 4.6. For now, based on the manual analysis procedure described, we only state the hypothesis that ACCM and AMLOC metrics are different between autonomous agents software and general software in other domains. Moreover, it seems robotics code lies somewhere in between, in terms of these metrics.

4.5 Machine Learning Analysis

A second approach for our investigation uses machine learning techniques, to complement the manual analysis. Humans detect patterns in visualizations that computers may miss, yet may also fall prey to misconceptions. Thus an automated analysis can complement the manual process.

We attempted to use several different machine learning classifiers to distinguish agent and non-agent software domains, with the goal of analyzing successful classification schemes, to reveal the metrics, or metric combinations, which prove meaningful in the classification.

Pre-processing the data. We filtered outliers at the top and bottom 3% of the data (i.e., within the 3–97 percentiles). Aggregated features (total, median, etc.) were removed to minimize the effect of project size on the model, and to reduce the number of features (standing originally at around 250). The data was divided into a training (85%) and testing (15%) sets.

Classification procedure. We choose one vs many classification strategy, similarly to the manual analysis above. Iterating over all software classes, we trained a binary classifier to differentiate between samples of one software domain (ex. Audio) to all other software classes. This creates an inherent imbalance in the number of examples presented, which we alleviated by using random over-sampling of the minority class.

For classification, we used the following classification algorithms: *Support Vector Machines*, *Logistics Regression*, and *Gradient-Boosted Decision Trees*. The implementations are open-source packages (scikit-learn[4] and XGBoost[5]). The performance of classifiers was carried out using two scoring functions, familiar to machine learning practitioners: F1 and AUC (area under the ROC curve). In both, a greater value indicates better performance. Each of the tables below (Tables 4.4–4.6) shows the top classifiers

[4]https://scikit-learn.org/.
[5]https://github.com/dmlc/xgboost.

Table 4.4 SVM top five scoring software domain classifiers.

Class	Repositories	F1 Score	Feature Ranking
Agent	Robocup-2D	0.67	[accm_quantile_median, accm_quantile_upper, noc_mean]
Agent	ANAC	0.62	[accm_quantile_lower, noa_quantile_lower, noc_mean]
General	IDE	0.33	[acc_quantile_lower, dit_mean, noc_quantile_upper]
General	Mobile	0.32	[acc_quantile_lower, acc_quantile_median, accm_quantile_lower]
General	Graphics	0.20	[accm_quantile_lower, dit_quantile_lower, dit_quantile_median]

Table 4.5 Logistic Regression top scoring software classes.

Class	Repositories	AUC	F1	Feature Ranking
Agent	ANAC	0.99	0.80	[accm_quantile_upper, noa_quantile_lower, rfc_quantile_lower]
Agent	Robocup-2D	0.97	0.70	[amloc_quantile_upper, noc_mean, npa_mean]
Robot	Robcup-Other-Leagues	0.87	0.50	[amloc_quantile_lower, cbo_mean, nom_mean]
General	IDE	0.77	0.44	[amloc_quantile_median, anpm_mean, npa_mean]
General	Graphics	0.76	0.24	[anpm_mean, lcom4_mean, mmloc_mean]

built using the classification algorithms. In each, we list the top classification results of a single domain versus all others. Our interest, however, is not so much on being able to classify a specific domain, but instead in the metrics used as features when classifying Agent software. The last column of each table lists the most informative 3–4 features (metrics) used by the classifier. Frequent recurrence may hint at important metrics.

Table 4.4 shows the top results from the SVM classifiers, in decreasing order of performance. SVM classification output is only "Hard decision" without probability distribution of the different classes and thus the AUC score is not available for it. We used the default SVM parameters in the implementation. The F1 scores in the table are far from indicating great success, yet we note the presence of the mean ACCM metric in the list of features important for classification for the repositories belongs to the "Agent" class.

We next used classifiers built using Logistic Regression (LR). We used L2 regularization, and stopping criteria of 100 iterations. The top LR classifiers are reported in Table 4.5. In general their scores are lower than the SVM classifiers reported above.

Finally, we used Gradient-Boosted Decision Tree classifiers. The idea in this technique is to use an ensemble of decision trees based on subsets of the samples and features, to lower the risk of over-fitting while maintaining high accuracy. The classifiers were built using the XGBoost package, using the default parameters. The results are shown in Table 4.6. Overall, the results are much better than the other two classification attempts. Some individual domain classifiers achieve high scores.

Table 4.6 Gradient Boosted Decision Trees top scoring software classes, in decreasing order of F1 scores.

Agent/General	Class (Domain)	AUC	F1
Agent	Robocup-2D	0.97	0.85
Agent	ANAC	0.98	0.67
Agent	Chess	0.84	0.44
Robot	Robcup-Other-Leagues	0.89	0.40
General	Graphics	0.65	0.31
General	Security	0.76	0.27
General	Mobile	0.80	0.22
General	Games	0.49	0.00
General	Audio	0.56	0.00
General	Robot-Simulation	0.66	0.00
General	Education	0.66	0.00
General	Finance	0.73	0.00
General	IDE	0.75	0.00
Robot	Robo-Projects	0.86	0.00

Most importantly, however, we note that the top performing classifiers (1) are those that are able to distinguish agent software from other types of software, and (2) utilize the mean ACCM and AMLOC metrics in their classification decisions. These results concur with the conclusions of the manual analysis described earlier. We also observe that software from physical robots participating in RoboCup (domain: *Robocup-Other-Leagues*) has also been classified successfully, using the AMLOC metric (among other metrics).

4.6 Discussion

Ultimately, our goal in this investigation is not only finding out if there is a difference between agent or robot software, and other software domains, but also to uncover the nature of this difference. This section discusses the results presented above, and attempts to draw conclusions, lessons, and hypotheses for future investigations.

ACCM and Control Complexity of Agents. First, it is clear that the ACCM measure is a recurring metric in successful classification schemes distinguishing agent software from other software. This is true both in the manual analysis, as well as in classifiers generated by machine learning algorithms. In general, Agent software seems to have high ACCM measurements, compared to other software domains. Robot software does have

higher ACCM (on average) than non-agent software, but the difference is much less pronounced than between Agent and general software. It is therefore immediately interesting to better understand what the ACCM actually measures.

ACCM — Average Cyclomatic Complexity per Method — is a more modern variant of the Cyclomatic Complexity (CC) metric introduced by McCabe in 1976 [24]. Briefly, the cyclomatic complexity of software is a measure of the number of possible execution paths through its control flow graph. The more branching points, conditional loops, and decision points in the software, the greater its CC. The ACCM measures the CC value at the method level, for all methods within a module. It then computes the mean of these measurements to introduce a single value which represents the complexity of the module as a whole.

Cyclomatic Complexity has been generally shown to be inversely correlated to code quality and defect frequency. Greater CC is correlated with a greater number of defects in the software, persistent bugs, and other indications of poor design and code quality. Indeed, the correlation is sufficiently accepted, that there exists recommended practices for the maintenance of CC values of new software within accepted *safe range*, below the ACCM measurements we generally see here.

Is Agent Software Inherently More Complex? (In short: YES!) There are alternative explanations for the higher ACCM values we observe in agent software: (1) that agent software is just inherently more complex, because the tasks tackled by the software *requires* greater complexity in the control flow of the software. Or, (2) that the agent code is just more buggy, or written by programmers who are not as well-trained, e.g., too academic?

We offer evidence that the first explanation is the correct one, i.e., that agent software is inherently more complex. One benefit of using competition software in this study is that alongside the software metrics, we also have clear quality metrics in terms of the success of the software. Specifically, we show below (Fig. 4.6) a plot of the ACCM measure from a subset of RoboCup software agent, vs the code effectiveness as measured by the mean goal difference of the agents in competitions. We see a clear inverse relation between the two: higher ACCM is associated with poor performance, just as it is in other software domains. However, the ACCM of *winning* agents is still higher than standard practice in software.

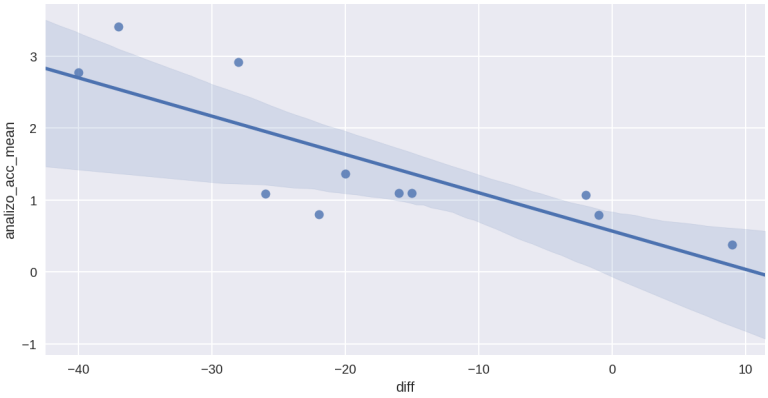


Fig. 4.6 ACCM (Average Cyclomatic Complexity per Module, vertical axis) vs Effectiveness (here, measured by *mean goal difference per game* — horizontal axis, larger is better). The goal difference was extracted automatically from log files of individual games.

What about other measures? A critical look at the results of this study raises the issue of other measures. It is true that ACCM is a clear distinguishing characteristic of agent vs non-agent code. However, it is not so clear that the machine learning classifiers can use it, ignoring other metrics. Indeed, some very successful classifiers do not use ACCM at all. Indeed, we saw also that the AMLOC measure is also a potentially good metric from this respect, as well as the MMLOC measure.

While we do not refute the possibility that other metrics may be as good as ACCM or complement it, we point out that many metrics are known to be correlated in practice (see, e.g., [41]), and thus it may be that a machine learning classifier using a particular metric could have also worked as well with a different one, that is highly correlated. In particular, in our own study here, we found that the Pearson correlation between AMLOC and ACCM is 0.84, and the correlation between MMLOC and ACCM is 0.90. So a preference for one metric over another does not necessarily mean that the other metric was not as useful.

4.7 The Big Picture and Future Directions

This paper offers the first empirical evidence that agent software is indeed inherently different from other types of software, intended for other domains. The empirical evidence was collected by analyzing hundreds of

software projects of comparable sizes, using two different types of analysis. In particular, we find that *agent software has greater control flow complexity* in general, which conjecture to be inherent to the types of tasks agents are deployed to solve — tasks that require autonomy in decision-making, and thus careful deliberation over many possibilities.

Given this conclusion, it becomes clear that agent-oriented software engineering can increase their impact by providing tools, methodologies, and frameworks that directly tackle the issue of complexity. For instance, agent architectures may be so successful because they assist in breaking down the inherent complexity of tasks. We leave this question for future work.

References

[1] Y. Shoham, Agent-oriented programming, *Artif. Intell.* **60**, 1, pp. 51–92 (1993), http://dx.doi.org/10.1016/0004-3702(93)90034-9.

[2] L. Padgham, J. Thangarajah and M. Winikoff, Prometheus Research Directions, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 155–171 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_8.

[3] S. A. DeLoach, O-MaSE: An Extensible Methodology for Multi-agent Systems, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 173–191 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_9.

[4] J. J. Gomez-Sanz, Ten Years of the INGENIAS Methodology, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 193–209 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_10.

[5] O. Boissier, R. H. Bordini, J. F. Hübner and A. Ricci, Unravelling Multi-agent-Oriented Programming, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 259–272 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_13.

[6] N. R. Jennings, On agent-based software engineering, *Artificial Intelligence* **117**, 2, pp. 277–296 (2000-03-01), http://www.sciencedirect.com/science/article/pii/S0004370299001071.

[7] A. Sturm and O. Shehory, Agent-Oriented Software Engineering: Revisiting the State of the Art, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 13–26 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_2.

[8] E. Platon, N. Sabouret and S. Honiden, An architecture for exception management in multiagent systems, *International Journal of Agent-Oriented Software Engineering* **2**, 3, p. 267 (2008), http://www.inderscience.com/link.php?id=19420.

[9] M. Winikoff, Future Directions for Agent-Based Software Engineering, *Int. J. Agent-Oriented Softw. Eng.* **3**, 4, pp. 402–410 (2009), http://dx.doi.org/10.1504/IJAOSE.2009.025319.

[10] B. P. Gerkey, R. T. Vaughan and A. Howard, The player/stage project: Tools for multi-robot and distributed sensor systems, in *Proceedings of the International Conference on Advanced Robotics* (2003), http://cres.usc.edu/cgi-bin/print_pub_details.pl?pubid=288.

[11] R. T. Vaughan and B. P. Gerkey, Really Reusable Robot Code and the Player/Stage Project, in Brugali, D. (ed.), *Software Engineering for Experimental Robotics*, p. 24 (2006).

[12] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng, ROS: an open-source Robot Operating System, in *International Conference on Robotics and Automation*, p. 6 (2009).

[13] D. Brugali, *Software Engineering for Experimental Robotics*. Springer (2007), ISBN 978-3-540-68951-5, google-Books-ID: DEpsCQAAQBAJ.

[14] D. Brugali and P. Scandurra, Component-based robotic engineering (Part I), *IEEE Robotics Automation Magazine* **16**, 4, pp. 84–96 (2009).

[15] D. Brugali and A. Shakhimardanov, Component-Based Robotic Engineering (Part II), *IEEE Robotics Automation Magazine* **17**, 1, pp. 100–112 (2010).

[16] D. Brugali, Model-Driven Software Engineering in Robotics: Models Are Designed to Use the Relevant Things, Thereby Reducing the Complexity and Cost in the Field of Robotics, *IEEE Robotics Automation Magazine* **22**, 3, pp. 155–166 (2015).

[17] D. Calisi, A. Censi, L. Iocchi and D. Nardi, Openrdk: A modular framework for robotic software development, in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1872–1877 (2008).

[18] A. Elkady and T. Sobh, Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography, *Journal of Robotics* **2012**, pp. 1–15 (2012), http://www.hindawi.com/journals/jr/2012/959013/.

[19] E. Tsardoulias and P. Mitkas, Robotic frameworks, architectures and middleware comparison, *arXiv:1711.06842 [cs]* (2017), http://arxiv.org/abs/1711.06842, arXiv: 1711.06842.

[20] M. Montemerlo, N. Roy and S. Thrun, Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit, in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No 03CH37453)*, Vol. 3, pp. 2436–2441 (2003).

[21] N. T. Dantam, K. B⊘ondergaard, M. A. Johansson, T. Furuholm and L. E. Kavraki, Unix Philosophy and the Real World: Control Software for Humanoid Robots, *Frontiers in Robotics and AI* **3** (2016).

[22] H. Bruyninckx, Open robot control software: the OROCOS project, in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, Vol. 3, pp. 2523–2528 (2001).

[23] N. T. Dantam and M. Stilman, Ach: IPC for Real-Time Robot Control, Technical Report GT-GOLEM-2011-001, Georgia Institute of Technology (2011).

[24] T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* **SE-2**, 4, pp. 308–320 (1976).

[25] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc. (1977), ISBN 978-0-444-00205-1.

[26] A. J. Albrecht, Measuring application development productivity, in *IBM Applications Development Joint SHARE/GUIDE Symposium*. Monterey, California, pp. 83–92 (1979).

[27] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd edn. McGraw-Hill, New York (2008).

[28] B. W. Boehm, *Software Engineering Economics*, 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1981), ISBN 0138221227.

[29] S. R. Chidamber and C. F. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* **20**, 6, pp. 476–493 (1994-06).

[30] R. V. Hudli, C. L. Hoskins and A. V. Hudli, Software metrics for object-oriented designs, in *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 492–495 (1994-10).

[31] E. K. Piveta, A. Moreira, M. S. Pimenta, J. Araújo, P. Guerreiro and R. T. Price, An empirical study of aspect-oriented metrics, *Science of Computer Programming* **78**, 1, pp. 117–144 (2012-11), `http://linkinghub.elsevier.com/retrieve/pii/S0167642312000287`.

[32] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press (2014-10-01), ISBN 978-1-4398-3823-5, google-Books-ID: lx_OBQAAQBAJ.

[33] R. V. Kumar and R. Chandrasekaran, Classification of software projects using k-means, discriminant analysis and artificial neural network, *International Journal of Scientific & Engineering Research* **4**, 2, p. 7 (2013).

[34] L. B. L. De Souza and M. D. A. Maia, Do software categories impact coupling metrics? in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13. IEEE Press, ISBN 978-1-4673-2936-1, pp. 217–220 (2013), ISBN 978-1-4673-2936-1, `http://dl.acm.org/citation.cfm?id=2487085.2487128`.

[35] M. Stojkovski, *Thresholds for Software Quality Metrics in Open Source Android Projects*, Master's thesis, NTNU (2017).

[36] P. Meirelles, C. Santos Jr., J. Miranda, F. Kon, A. Terceiro and C. Chavez, A study of the relationships between source code metrics and attractiveness in free software projects, in *2010 Brazilian Symposium on Software Engineering*. IEEE, ISBN 978-1-4244-8917-6, pp. 11–20 (2010), ISBN 978-1-4244-8917-6, `http://ieeexplore.ieee.org/document/5631691/`.

[37] I. García-Magariño, M. Cossentino and V. Seidita, A Metrics Suite for Evaluating Agent-oriented Architectures, in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10. ACM, New York, NY,

USA, ISBN 978-1-60558-639-7, pp. 912–919 (2010), ISBN 978-1-60558-639-7, doi:10.1145/1774088.1774278, http://doi.acm.org/10.1145/1774088.1774278, event-place: Sierre, Switzerland.

[38] F. Alonso, J. L. Fuertes, L. Martínez and H. Soza, Measuring the Pro-Activity of Software Agents, *2010 Fifth International Conference on Software Engineering Advances*, pp. 319–324 (2010), doi:10.1109/ICSEA.2010.55.

[39] F. Alonso, J. L. Fuertes, L. Martinez and H. Soza, Towards a set of Measures for Evaluating Software Agent Autonomy, in *2009 Eighth Mexican International Conference on Artificial Intelligence*, pp. 73–78 (2009), doi:10.1109/MICAI.2009.15.

[40] M. Cossentino, C. Lodato, S. Lopes, P. Ribino and V. Palermo, Metrics for Evaluating Modularity and Extensibility in HMAS Systems, in *AAMAS* (2015).

[41] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft and C. Ward, Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship, *Journal of Software Engineering and Applications* **02**, 3, pp. 137–143 (2009), http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jsea.2009.23020.

AI for Software Development

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0005

Chapter 5

Sequence-to-Sequence Learning for Automated Software Artifact Generation

Zhongxin Liu[a], Xin Xia[b] and David Lo[c]
[a]*Zhejiang University, China*
[b]*Monash University, Australia*
[c]*Singapore Management University, Singapore*

5.1 Introduction

During the development and maintenance of a software system, developers produce many digital artifacts besides source code, e.g., requirement documents, code comments, change history, bug reports, etc. Such artifacts are valuable for developers to understand and maintain the software system. However, creating software artifacts can be burdensome and developers sometimes neglect to write and maintain important artifacts [1, 2]. This problem can be alleviated by software artifact generation tools, which can assist developers in creating software artifacts and automatically generate artifacts to replace existing empty ones.

There have been many approaches proposed to automatically generate software artifacts. Some of them rely on manually derived templates and rules for generation, namely, templated-based approaches [3–5]. Some other approaches generate software artifacts by leveraging Information Retrieval (IR) techniques to retrieve and reuse existing artifacts, i.e., IR-based approaches [6–8]. Recently, researchers proposed to adopt sequence-to-sequence (seq2seq) learning to automate software artifact generation. For example, there were many studies focusing on generating summaries for code snippets, i.e., code summarization, by learning from code-comment pairs using seq2seq models [9–12]. Some work adopted seq2seq models to

"translate" code changes to their corresponding commit messages [13, 14]. Several researchers also investigated the effectiveness of seq2seq models in generating pull request descriptions [2] and app review responses [15].

The focus of this chapter is automated software artifact generation (hereon, SAG) using seq2seq learning. This research direction is inspired by the similarities between natural language generation (NLG) and SAG and the effectiveness of seq2seq models on NLG tasks. When applied to SAG, seq2seq models are able to automatically learn generation patterns from massive software artifact data and adaptively adopt such learned patterns for generation. Compared to template-based and IR-based techniques, seq2seq-model-based approaches do not require expensive manual efforts to summarize and implement templates or rules, are not limited to term-based summaries, are able to produce novel expressions and can be more general. In addition, seq2seq learning is developing rapidly and there are more and more publicly available software artifacts on the Internet, which make seq2seq-model-based SAG a timely and promising research direction.

This chapter aims to provide a comprehensive introduction to this research direction, i.e., seq2seq-model-based SAG. Specifically, we first introduce the preliminary knowledge of seq2seq models, including the RNN, the encoder-decoder model, the attention mechanism, and some commonly-used evaluation metrics for SAG (Sec. 5.2). Next, three case studies, i.e., code comment generation, pull request description generation, and app review response generation, are presented to illustrate how to build SE-task-specific parallel corpora for seq2seq models and how to customize seq2seq models in a SE-task-specific way (Secs. 5.3–5.5). These case studies cover three important types of software artifacts, i.e., documents of source code, documents of code changes and responses for user feedback. Each of them presents the state-of-the-art result in a specific task. Then, we summarize the features of the tasks covered in the three case studies and make a comparison of the cases with respect to such features (Sec. 5.6). Finally, the challenges and opportunities of this research direction are discussed in Sec. 5.7 and we conclude this chapter in Sec. 5.8. This chapter can serve as a starting point for researchers and practitioners interested in this research area. Hopefully, it can inspire experts in this area to propose more advanced seq2seq models for generating diverse software artifacts and encourage and guide practitioners to apply these generative models in practice.

5.2 Preliminary Knowledge of Seq2Seq Learning

Attentional RNN-based encoder-decoder models are an effective and popular family of seq2seq models. Much work on SAG is based on such models.

5.2.1 *Recurrent Neural Network (RNN)*

The recurrent neural network (RNN) is a kind of neural network specialized for processing sequential data [16, 17]. It recurrently uses the same cell to process each element of a sequence, hence it can efficiently handle variable-length sequences.

To process a sequence of tokens $\boldsymbol{x} = \{x_1, x_2, ..., x_{|\boldsymbol{x}|}\}$, we first need to map each token x_i into a real-value vector $\boldsymbol{e}_i$ (usually through a learnable lookup table named *embedding layer*). Then at each time step i, the RNN takes as input $\boldsymbol{e}_i$ and the previous hidden state $\boldsymbol{h}_{i-1}$ to compute the current hidden state $\boldsymbol{h}_i$ as follows:

$$\boldsymbol{h}_i = f(\boldsymbol{h}_{i-1}, \boldsymbol{e}_i) \tag{5.1}$$

Long short-term memory (LSTM) [18] and gated recurrent unit (GRU) [19] are two popular implementations of f, which are capable of preserving long-term dependencies in sequences through gating mechanisms. $\boldsymbol{h}_i$ is expected to adaptively capture the information of x_1 to x_i, and can be regarded as x_i's representation.

Some tasks, such as part-of-speech tagging in NLP, require x_i's representation to capture information from the whole sequence instead of only the subsequence before x_i. Bidirectional RNNs [20] were invented to meet this need, which calculate not only the forward state $\boldsymbol{h}_i$ from left to right, but also a backward state $\boldsymbol{g}_i$ from right to left, as follows:

$$\boldsymbol{g}_i = f'(\boldsymbol{g}_{i+1}, \boldsymbol{e}_i)$$

f' is another RNN cell. The hidden state $\boldsymbol{h}_i^b$ of a bidirectional RNN at time step i is then constructed by concatenating $\boldsymbol{h}_i$ and $\boldsymbol{g}_i$:

$$\boldsymbol{h}_i^b = [\boldsymbol{h}_i : \boldsymbol{g}_i] \tag{5.2}$$

5.2.2 *Encoder-Decoder Model*

The encoder-decoder model can be used to "translate" a source sequence to a target sequence. It consists of two components, namely encoder and decoder, which are usually two distinct RNNs. The encoder is responsible for encoding the source sequence: It first maps each source token x_i into a

vector $\boldsymbol{e_i}$ using an embedding layer. At each encoding step i, x_i's hidden state $\boldsymbol{h}_i$ is calculated using Eq. (5.1) (RNN) or Eq. (5.2) (Bidirectional RNN). The last hidden state $\boldsymbol{h}_{|\boldsymbol{x}|}$, which can be regarded as the intermediate representation of the source sequence, is input into the decoder as its initial hidden state $\boldsymbol{s}_0$.

The decoder is trained to generate the target sequence by sequentially predicting each target token y_i conditioned on the current hidden state $\boldsymbol{s}_t$ and an input token $\hat{y}_{t-1}$:

$$p(y_t|y_{<t}, \boldsymbol{x}) = g(\hat{y}_{t-1}, \boldsymbol{s}_t) \tag{5.3}$$

where $\hat{y}_{t-1}$ is the previous reference token y_{t-1} when training or the previously generated token when testing. Specifically, $\hat{y}_{t-1}$ is first mapped to its embedding $\hat{\boldsymbol{e}}_t$. Next, the hidden state $\boldsymbol{s}_t$ is computed using Eq. (5.1) with $\boldsymbol{s}_{t-1}$ and $\hat{\boldsymbol{e}}_t$ as input. Then, $\boldsymbol{s}_t$ is projected to the vocabulary distribution P_{vocab} of this step as follows:

$$P_{\text{vocab}} = \text{softmax}(\boldsymbol{W}_v \boldsymbol{s}_t + \boldsymbol{b}) \tag{5.4}$$

where $\boldsymbol{W}_v$ and $\boldsymbol{b}$ are learnable parameters. P_{vocab} is used to decide which token to generate, which means:

$$p(y_t|y_{<t}, \boldsymbol{x}) = P_{\text{vocab}}(y_t) \tag{5.5}$$

The probability of generating $\boldsymbol{y}$ given $\boldsymbol{x}$ is hence calculated as:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \prod_{t=1}^{|\boldsymbol{y}|} p(y_t|y_{<t}, \boldsymbol{x})$$

The negative log-likelihood function is commonly used as the loss function and is defined as:

$$loss_{\text{nll}} = -\frac{1}{N} \sum_{n=1}^{N} log(p(\boldsymbol{y}|\boldsymbol{x})) \tag{5.6}$$

where N is the number of training samples. The parameters in the model can be estimated by minimizing $loss_{\text{nll}}$ through an optimization algorithm such as stochastic gradient descent (SGD).

5.2.3 *Attention Mechanism*

One fixed-dimension vector $\boldsymbol{h}_{|\boldsymbol{x}|}$ may not be powerful enough to capture all relevant information in a source sequence. Due to this problem, attention mechanisms were proposed to enhance the encoder-decoder model [21]. The goal of attention mechanisms is to adaptively derive a context vector $\boldsymbol{c}_t$

from encoder's hidden states at each decoding step t so that the decoder can better capture source-side information.

A commonly used attention mechanism named *additive attention* [21] works as follows: at each decoding step t, it takes as input all encoder hidden states $\{\boldsymbol{h}_1, ..., \boldsymbol{h}_{|x|}\}$ and the previous decoder hidden state $\boldsymbol{s}_{t-1}$, and computes $\boldsymbol{c}_t$ as the weighted sum of $\{\boldsymbol{h}_1, ..., \boldsymbol{h}_{|\boldsymbol{x}|}\}$:

$$\boldsymbol{c}_t = \sum_{i=1}^{|\boldsymbol{x}|} \alpha_{ti} \boldsymbol{h}_i \tag{5.7}$$

The weight α_{ti} is defined as:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{|\boldsymbol{x}|} \exp(e_{tk})} \tag{5.8}$$

where e_{ti} can be regarded as the importance score of x_i with respect to the current output token and is computed by an *alignment model*:

$$e_{ti} = \boldsymbol{v}_e^\top tanh(\boldsymbol{W}_e \boldsymbol{s}_{t-1} + \boldsymbol{U}_e \boldsymbol{h}_i) \tag{5.9}$$

where $\boldsymbol{v}_e$, $\boldsymbol{W}_e$ and $\boldsymbol{U}_e$ are the weight matrices jointly learned with the model.

Figure 5.1 presents the structure of the attentional encoder-decoder model. In such a model, $\boldsymbol{c}_t$ is also used to calculate the probability of generating y_t, so Eq. (5.3) is modified as:

$$p(y_t | y_{<t}, \boldsymbol{x}) = g(\hat{y}_{t-1}, \boldsymbol{s}_t, \boldsymbol{c}_t)$$

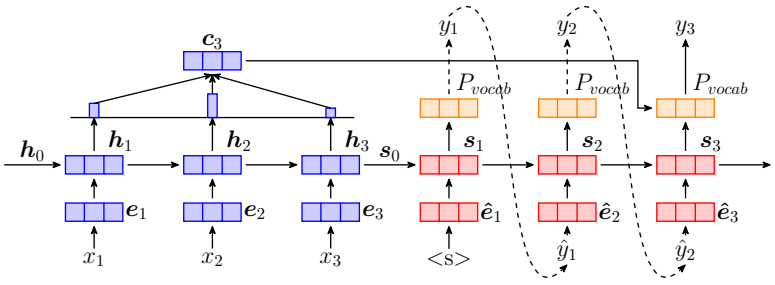


Fig. 5.1 **The structure of the attentional encoder-decoder model.** The encoder and decoder use two distinct RNNs. x_i and y_i refer to the i_{th} source and target tokens, respectively. $\boldsymbol{e}_i$ and $\hat{\boldsymbol{e}}_i$ are word embeddings. The representation $\boldsymbol{h}_i$ of x_i is calculated by the encoder. At decoding step t (this figure presents the snapshot where $t = 3$), the decoder takes as input the previous target token $\hat{y}_{t-1}$, calculates the hidden state $\boldsymbol{s}_t$ and the context vector $\boldsymbol{c}_t$, and then predicts the current target token y_t based on them.

5.2.4 *Evaluation Metrics for Software Artifact Generation*

Though accurate and convincing, human evaluation is expensive and time-consuming for evaluating SAG systems. For quick and automatic evaluation, researchers borrowed the automatic evaluation metrics designed for natural language generation tasks [9, 13, 22]. These metrics are quick, inexpensive, reasonably accurate, and shown to be highly correlated with human judgments. Three metrics are widely used to evaluate SAG systems, i.e., BLEU [23], METEOR [24, 25] and ROUGE [26].

BLEU is one of the de facto standards for machine translation evaluation. It measures the similarity of a hypothesis h (in our case, a generated artifact) and its corresponding (one or more) references R using an average modified n-gram precision with a penalty for overly short sentences. Specifically, given a corpus T with a number of hypothesis-references pairs, the (corpus-level) modified n-gram precision is defined as:

$$p_n = \frac{\sum_{(h,R)\in T} \sum_{\mathrm{ng}\in h} \mathrm{C}_{\mathrm{clip}}(\mathrm{ng}, h, R)}{\sum_{(h',R')\in T} \sum_{\mathrm{ng}'\in h'} \mathrm{C}(\mathrm{ng}', h')}$$

where ng is an n-gram, $\mathrm{C}(\mathrm{ng}, h)$ calculates ng's count in h and $\mathrm{C}_{\mathrm{clip}}(\mathrm{ng}, h, R)$ is defined as follows:

$$\mathrm{C}_{\mathrm{clip}}(\mathrm{ng}, h, R) = \min(C(\mathrm{ng}, h), \max_i C(\mathrm{ng}, R_i))$$

The BLEU score on T is then calculated as follows:

$$\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

N is the maximum n-gram order. w_n is the weight of p_n. BP is the brevity penalty factor, defined as:

$$\mathrm{BP} = \begin{cases} 1 & \text{if } l_h > l_r \\ e^{1-l_r/l_h} & \text{if } l_h \leq l_r \end{cases}$$

where l_h is the sum of the lengths of all hypotheses and l_r is calculated by summing the lengths of each hypothesis's best-matched reference. In practice, w_n is usually set to $1/N$. N is usually set to 4, and the corresponding BLEU score is referred to as BLEU-4 score. In SE, BLEU has been used to evaluate code comment generation systems [9, 10, 12, 27, 28], commit message generation methods [8, 13, 14], API sequence generation approaches [22], etc.

METEOR is another popular evaluation metric for machine translation. Different from BLEU, METEOR leverages multiple matchers, including exact, stem, synonym and paraphrase matchers, to accurately match words and phrases between a hypothesis and its corresponding reference. Also, it considers recall by calculating an F-score and combines the F-score with a fragmentation penalty to account for gaps and differences in word order. Researchers have used it to evaluate code comment generation methods [10, 12, 28] in SE.

ROUGE is a set of metrics proposed by [26] to evaluate text summarization systems. Similar to BLEU, ROUGE relies on co-occurrences of n-grams, word sequences and word pairs between hypotheses and references. But it values recall more because a text summarization system is required to generate or retrieve as much important information as possible. ROUGE-N (N = 1, 2) and ROUGE-L are the most commonly used ROUGE metrics, which are based on n-gram co-occurrences and longest common subsequences, respectively. Some work in SE leveraged ROUGE to assess the quality of generated code comments [12] and generated pull request descriptions [2]. For the specific formulas of METEOR and ROUGE, we refer readers to the papers of METEOR and ROUGE.

5.3 Automated Generation of Code Comments

Good comments can improve the efficiency of program comprehension activities [4] and hence can save developers' time. However, writing comments can be cumbersome for developers, especially when the schedule is tight. Therefore, comments are often mismatched, missing and outdated in some projects.

Many code comment generation approaches are proposed to alleviate this problem, which can be divided into three categories. The first category of work generates comments using manually defined templates [4, 29–31]. For example, [4] proposed to generate Java method comments based on the Software Word Usage Model (SWUM) and manually-crafted templates. Another line of research leverages Information Retrieval (IR) techniques for generation [6, 7, 32]. For instance, ColCom, a tool proposed by [7] generates comments for code snippets based on code clone detection techniques. These approaches are promising but have two main limitations: 1) relying on well-named identifiers in source code. 2) requiring the existence of similar code snippets, which is not always true.

Recently, the third category of approaches, which uses probabilistic

models to generate comments, emerges and attracts a lot of attention [9, 10, 12, 27, 28, 33]. Different from rule-based and IR-based methods, these approaches first train generative models on code-comment pairs, and then directly predict comments from code using trained models. One of the early work in this category is CODE-NN [27], which jointly performs content selection and surface realization, i.e., comment generation, through an attentional RNN model.

In this section, we describe an approach named Hybrid-DeepCom [10], which is one of the state-of-the-art seq2seq-model-based techniques for generating code comments, as a case study.

5.3.1 *Problem and Challenges*

Hybrid-DeepCom aims to automatically generate high-level descriptions for Java methods. Formally, given a Java method $\boldsymbol{x}$ and its corresponding high-level description $\boldsymbol{y}$, the aim is to find a function f so that $\boldsymbol{y} = f(\boldsymbol{x})$. [10] regarded this problem as a machine translation problem based on the insight that the transition between source code and comments is similar to that between different natural languages. Naturally, as the state-of-the-art methods for machine translation, seq2seq models were adopted by them to solve this problem. However, to model source code using seq2seq models, several challenges need to be addressed:

(1) How to take advantage of the syntactic information of source code? Natural language text is weakly structured. In contrast, programming languages are formal languages and source code written in them are unambiguous and structured [34]. Therefore, considering syntactic information may boost the effectiveness of seq2seq models on source code.
(2) How to fuse the lexical information and the syntactic information? On the one hand, comment tokens may be extracted or distilled from code identifiers. On the other hand, the syntactic structure of code may help seq2seq models capture the semantic information of code and generate better comments. So, the lexical and the syntactic information should be combined in a proper way to make seq2seq models more effective.
(3) How to deal with the out-of-vocabulary (OOV) tokens in source code? Seq2seq models model and generate sequences based on the vocabulary of sequence tokens. The tokens not included in the given vocabulary, i.e., out-of-vocabulary (OOV) tokens, cannot be properly handled. In code corpus, developer-named identifiers are pervasive and may result

in many OOV tokens if the vocabulary size is limited. This issue should be properly addressed to guarantee the effectiveness of seq2seq models on code corpus.

5.3.2 *The Seq2Seq Model: Hybrid-DeepCom*

The framework of Hybrid-DeepCom is presented in Fig. 5.2. Hybrid-DeepCom is based on the attentional encoder-decoder model we mentioned in Sec. 5.2.2 and Sec. 5.2.3, but enhances it in several ways to handle the challenges mentioned above. First, it uses two encoders, namely code encoder and AST encoder, to encode code tokens and AST nodes, respectively. Second, it customizes the attention mechanism to adaptively fuse the lexical information and the syntactic information extracted by the two encoders. Third, Hybrid-DeepCom reduces the number of OOV tokens by splitting identifiers into common words.

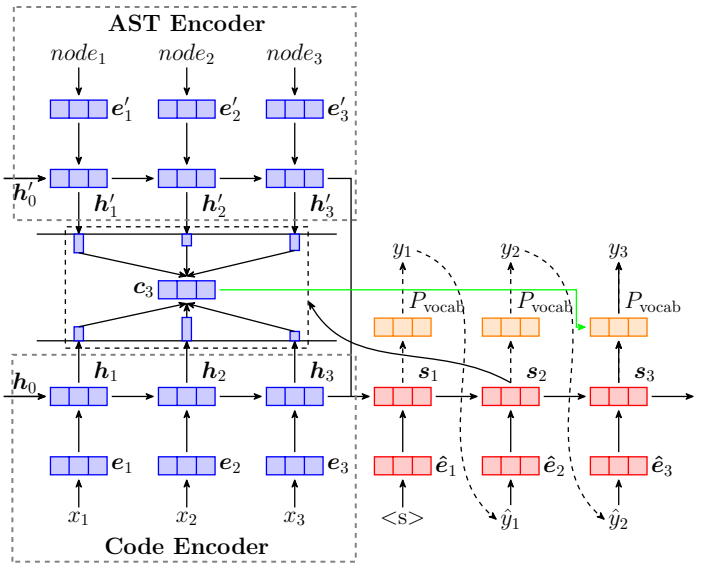


Fig. 5.2 **The framework of Hybrid-DeepCom.** The differences between Hybrid-DeepCom and the basic attentional encoder-decoder model are: First, Hybrid-DeepCom leverages two encoders to represent code tokens and AST nodes, respectively. Second, the decoder calculates two context vectors based on the output of the two encoders and concatenates them as the final context vector c_i. Third, identifiers are split into common words before being processed. $node_i$ refers to the i_{th} element of the node sequence output by the SBT method.

5.3.2.1 *Capture Syntactic Information of Source Code*

We can process a code snippet using a seq2seq model by simply representing the code as a sequence of tokens. However, the model may miss capturing the structural information of the code in this way. Source code can also be parsed into ASTs, which explicitly present the syntactic information. To help seq2seq models capture such syntactic information, an intuitive way is converting ASTs into sequences of AST nodes and then directly inputting them into an encoder. However, node sequences produced by classic traversal methods, e.g., pre-order traversal, are lossy because they cannot be recovered to the original ASTs. The ambiguity of such sequences also means samples (in this case, Java methods) with different labels (in this case, comments) may be mapped to the same representation, which may confuse the neural network.

To tackle this problem, Hybrid-DeepCom proposes a Structural-Based Traversal method, namely SBT, to produce node sequences. Given a tree where each node i is of type T_i, the SBT method outputs its node sequence as follows: 1) It traverses the tree in pre-order. 2) Before visiting node i, it outputs "$(T_i$"; after visiting node i, it outputs "$)Ti$" to mark the end of node i. For example, the sequence produced for the tree in Fig. 5.3 is "$T_1((T_2)T_2(T_3(T_4)T_4(T_5)T_5(T_6)T_6)T_3)T_1$". As we can see, by leveraging a pair of brackets to mark the tree structure, SBT makes the node sequences lossless.

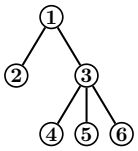


Fig. 5.3 **A simple tree for illustrating the Structural-Based Traversal (SBT) method**.

5.3.2.2 *Fuse Lexical and Syntactic Information*

Hybrid-DeepCom learns to capture lexical information from code token sequences using the code encoder and syntactic information from AST node sequences through the AST encoder. The hidden states of all encoders need to be projected into a shared space to calculate the distribution $p(y_t|y_{<t}, \boldsymbol{x})$. To meet this requirement, Hybrid-DeepCom computes a context vector for

each encoder according to Eqs. (5.7), (5.8) and (5.9). Then the final context vector is defined as the sum of the two vectors:

$$\boldsymbol{c}_t = \boldsymbol{c}_t^{\text{code}} + \boldsymbol{c}_t^{\text{ast}}$$

$\boldsymbol{c}_t$ is expected to contain both lexical and syntactic information of source code, hence it can better guide the generation of comments.

5.3.2.3 *Reduce OOV tokens*

To model and generate sequences, seq2seq models need to build the vocabulary of sequence tokens. Due to the pervasive developer-named identifiers in source code, a small vocabulary will result in a large number of OOV tokens and make it difficult for seq2seq models to distinguish different identifiers, while a large vocabulary requires more training data, time and computation memory. Based on the observation that most identifiers are composed of several subtokens that are usually common words, Hybrid-DeepCom alleviates this issue by splitting identifiers into several words. Consequently, the code encoder only needs to model common words instead of rare identifiers. As a result, this strategy reduced the number of unique tokens in source code by 90%.

5.3.3 *Dataset*

A good dataset is critical for a data-driven approach. To evaluate Hybrid-DeepCom, [10] built a parallel corpus of <method, comment> pairs as follows: They first collected the Java projects created from 2015 to 2016 with more than 10 stars from GitHub and obtained 9,714 projects. Then, they parsed these Java projects, extracting Java methods and the corresponding Javadocs. Only the first sentences of the Javadocs were kept as method summaries. To reduce noise, <method, comment> pairs of which the comments only contain one word or the methods are setters, getters, constructors or tests were removed. In addition, the overridden Java methods were also excluded since they usually implement the same functionality and may cause unwanted repetition. Furthermore, to better suit the seq2seq model, [10] set the maximum lengths of Java methods and comments as 200 and 30, respectively, based on the statistics of their collected data. Finally, [10] constructed a dataset with 485,812 <method, comment> pairs.

5.3.4 *Evaluation*

Hybrid-DeepCom was evaluated on the above-mentioned dataset in terms of BLEU, METEOR and some IR metrics. The BLEU score and the METEOR of Hybrid-DeepCom are reported as 34.51 and 24.46, which improve over CODE-NN by 61.9% and 35.6%, respectively. It can outperform the basic attentional encoder-decoder model by 15.6% and 11.0% in terms of the two metrics. Also, the human evaluation conducted by [10] confirmed the better performance of Hybrid-DeepCom compared to CODE-NN. In summary, by customizing the attentional encoder-decoder model to address task-specific challenges, Hybrid-DeepCom achieves better performance than the baselines on code comment generation.

5.4 Automated Generation of Pull Request Descriptions

After making a code change, developers can attach a free-form description to illustrate what was changed and/or why to change. Change descriptions can help developers gain quick and adequate understandings of code changes without digging into the details, hence are valuable for program comprehension and software maintenance tasks. However, similar to code comments, change descriptions sometimes are neglected by developers due to tight schedules [2]. To address this problem, some approaches have been proposed to automatically generate descriptions for code changes [2, 3, 13, 35].

According to granularity, code changes fall into three types: commits, pull requests and releases. Commits can be regarded as the basic unit of code changes. Researchers have proposed to automatically document commits based on diverse inputs [3, 8, 13, 36–39]. For example, [3] proposed a tool named DeltaDoc to generate commit messages based on symbolic execution and manually crafted templates. Seq2seq models were first adopted by [13] to automatically generate commit messages from corresponding diffs. A release is a large change with a number of commits and/or pull requests. Some work has also explored the automatic generation of release notes [35, 40]. As an example, [40] proposed a tool named ARENA, which combines several techniques and manually defined templates to generate release notes from code change history and other software artifacts, e.g., bug reports.

At present, the pull-based development model is very popular. As the communication medium between developers and project maintainers, pull

requests (PRs) are the key to this development model. A PR usually contains one or more interrelated commits and is usually far smaller than a release. In a recent work, [2] found that pull request descriptions (PR descriptions) sometimes are also ignored by developers. They proposed to regard the generation of PR descriptions as a text summarization problem and first attempted to generate PR descriptions through a seq2seq model. In this section, we introduce this work in detail. For convenience, we refer to the model proposed by [2] as PRSummarizer.

5.4.1 *Problem and Challenges*

[2] aimed to generate a PR description from the commits in the corresponding PR. They noticed that the information appearing in a PR description usually can be found in the commit messages and the added code comments in this PR. Therefore, they proposed to regard this task as a text summarization task, where the commit messages and the added code comments in a PR are combined as the "article" and the description of this PR is treated as the "summary". Considering the advances on text summarization tasks after using seq2seq models, [2] also chose to leverage a seq2seq model to generate "summaries" (the target sequence) from "articles" (the source sequence).

However, they found that the basic attentional encoder-decoder model is not enough for solving this problem because of two main challenges:

(1) Out-of-vocabulary (OOV) tokens. As mentioned in Sec. 5.3.1, OOV tokens are pervasive in source code. In fact, they are also common in software documents. Different from code summarization, this task cares more about the effective generation of OOV tokens than their representation.
(2) The gap between the training objective and the real quality of generated PR descriptions. The objective of a seq2seq model is usually minimizing the negative log-likelihood loss presented in Eq. (5.6). This loss function requires a generated text to be literally the same as the corresponding reference. However, two different texts may convey the same meaning. Thus, there are gaps between the commonly-used loss function and human judgments, which may make the trained model with the least loss not be the model with the best performance.

5.4.2 *The Seq2Seq Model: PRSummarizer*

The framework of PRSummarizer is presented in Fig. 5.4, which is also based on the attentional encoder-decoder model. A copy mechanism named pointer generator [41] is adopted to deal with the OOV tokens. To bridge the gap between the training objective and human evaluation, PRSummarizer directly optimizes ROUGE during training through a reinforcement learning (RL) technique.

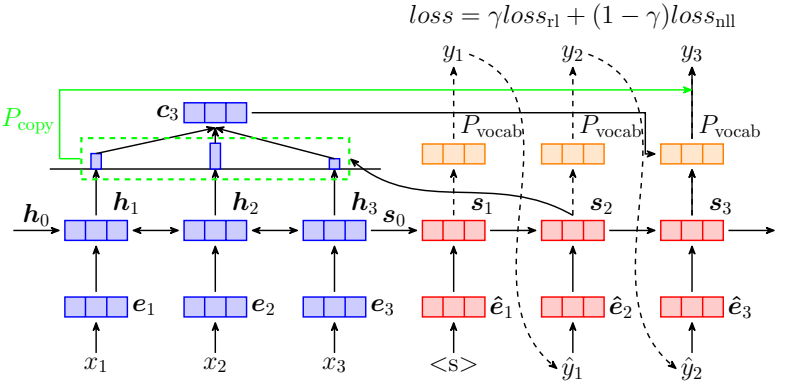


Fig. 5.4 **The framework of PRSummarizer**. Different from the basic attentional encoder-decoder model, PRSummarizer applies the pointer generator to handle OOV tokens by copying from the source sequence and uses a reinforcement-learning-based loss to better guide the optimization of the model. P_{copy} refers to the vocabulary distribution calculated by the pointer generator.

5.4.2.1 *Pointer Generator*

Based on the observation that the OOV tokens in a PR description can often be found in the "article", i.e., the commit messages and the added comments in this PR, PRSummarizer integrates the pointer generator with the attentional encoder-decoder model. With this component, the output tokens can be generated by selecting from the fixed vocabulary or "copying" from the source sequence.

In detail, at decoding step t, a *generation probability* is calculated based on the embedding $\hat{\boldsymbol{e}}_t$ of the current decoder input $\hat{y}_t$, the decoder state $\boldsymbol{s}_t$ and the context vector $\boldsymbol{c}_t$:

$$p_t^{\text{gen}} = \sigma(\boldsymbol{w}_c^\top \boldsymbol{c}_t + \boldsymbol{w}_s^\top \boldsymbol{s}_t + \boldsymbol{w}_e^\top \hat{\boldsymbol{e}}_t + b_{\text{gen}})$$

where $\boldsymbol{w}_c$, $\boldsymbol{w}_s$, $\boldsymbol{w}_e$ and b_{gen} are learnable parameters, and σ is the sigmoid function. p_t^{gen} is expected to measure the probability that y_t is generated from the fixed vocabulary. Therefore, the conditional probability of generating y_t is modified from Eq. (5.5) to:

$$p(y_t|y_{<t}, \boldsymbol{x}) = p_t^{\text{gen}} P_{\text{vocab}}(y_t) + (1 - p_t^{\text{gen}}) P_{\text{copy}}(y_t)$$

where $P_{\text{copy}}(y_t)$ is the probability of "copying" y_t from the source sequence and is computed based on the attention weights defined in Eq. (5.8):

$$P_{\text{copy}}(y_t) = \sum_{i|x_i=y_t} \alpha_{ti}$$

We can see that when y_t is an OOV token, $P_{\text{vocab}}(y_t)$ will be zero, but if y_t appears in the "article" $\boldsymbol{x}$, the decoder can also generate y_t according to $P_{\text{copy}}(y_t)$. In this way, OOV token issue can be alleviated.

5.4.2.2 *RL Loss*

Due to the gap between the training loss and human judgments, researchers usually use ROUGE, which is flexible and shown to highly correlate with human assessments, to evaluate the quality of generated summaries. An intuitive idea to bridge this gap is directly optimizing for ROUGE during training. However, ROUGE is discrete, which means the model's parameter gradients cannot be calculated from ROUGE scores. To tackle this issue, PRSummarizer leveraged a reinforcement learning technique named self-critical sequence training (SCST) [42] and adopted a special loss function named RL loss.

The generation of PR descriptions can be described using RL terminology. The encoder's output and the decoder's input compose the "environment". The decoder is the "agent" which takes an "action", i.e., generates a token, at each decoding step according to a "policy" π_θ. The policy is indeed the neural network of the decoder and θ is its parameters. ROUGE-L is used as the "reward" function. Therefore, after the decoder produces a PR description $\hat{\boldsymbol{y}}$, a reward can be calculated as follows:

$$r(\hat{\boldsymbol{y}}) = \textit{ROUGE-L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

where $\boldsymbol{y}$ is the ground truth. The objective of this RL problem is defined to maximize the expected reward, i.e., minimize the negative expectation:

$$L(\theta) = -\mathbb{E}_{\boldsymbol{y}^s \sim \pi_\theta}[r(\boldsymbol{y}^s)]$$

where $\boldsymbol{y}^s$ is a PR description sampled from π_θ.

According to the SCST algorithm, the expected gradient of $L(\theta)$ can be computed as:

$$\nabla_\theta L(\theta) = -((r(\boldsymbol{y}^s) - r(\boldsymbol{y}^b))\nabla_\theta \log p(\boldsymbol{y}^s|\boldsymbol{x})$$

where $\boldsymbol{y}^b$ is the baseline sequence generated by π_θ through greedy decoding, i.e., outputting the token with the highest probability at each step. Based on this gradient, a special loss named RL loss is defined as:

$$loss_{\text{rl}} = -\frac{1}{N}\sum_{n=1}^{N}((r(\boldsymbol{y}^s) - r(\boldsymbol{y}^b))\log p(\boldsymbol{y}^s|\boldsymbol{x})$$

where $r(\boldsymbol{y}^s)$ and $r(\boldsymbol{y}^b)$ are regarded as constants during gradient calculation. We can see that $loss_{\text{rl}}$ can be considered as a normalized version of $loss_{\text{nll}}$ (defined in Eq. (5.6)). If $\boldsymbol{y}^s$ is better than $\boldsymbol{y}^b$, i.e., $r(\boldsymbol{y}^s) - r(\boldsymbol{y}^b) > 0$, minimizing $loss_{\text{rl}}$ is equivalent to maximizing the probability of generating $\boldsymbol{y}^s$, and vice versa.

PRSummarizer used a hybrid training loss by combining the $loss_{\text{nll}}$ and the $loss_{\text{rl}}$ to bridge the above-mentioned gap without reducing the readability of generating PR descriptions, as follow:

$$loss = \gamma loss_{\text{rl}} + (1-\gamma)loss_{\text{nll}}$$

5.4.3 *Dataset*

To evaluate the effectiveness of PRSummarizer, [2] built a parallel corpus of which the source sequence is the commit messages and the added comments in a PR and the target sequence is the description of this PR. They first selected the top 1000 engineered Java projects from GitHub according to the number of merged PRs. For each project, at most 1000 merged PRs are downloaded. The description, commit messages and added comments in each merged PR are parsed and extracted. In total, 333,001 merged PRs are collected.

The preprocessing of PR data is troublesome since some texts in a PR are irrelevant to answering what was changed in this PR or why this change happened. To filter out such texts as many as possible, [2] applied a procedure $\mathcal{P}$ to carefully process all kinds of texts, i.e., the description, the commit messages and the added comments, in a PR. Given a text, this procedure first removes all HTML contents and "checklist" paragraph in it. The text is then split into sentences, and sentences with URLs, internal references (e.g., "#123"), signatures (e.g., "signed-off-by"), emails,

"@name" and markdown headlines are filtered out. Finally, the texts with more than 50% non-ASCII tokens are marked as "non-ASCII".

Give a PR, its commit messages were preprocessed using the procedure $\mathcal{P}$ and were combined as the commit message paragraph. As for the added comments in each commit, copyright comments, license comments, function signatures in Java docs and comments with only punctuation were deleted. The remaining comments are combined as the comment paragraph of this commit. Each comment paragraph is also preprocessed by the procedure $\mathcal{P}$. The commit message paragraph and the comment paragraphs are concatenated as the source sequence, or the "article".

The target sequence, or the "summary", only contains the PR description and was simply processed by the procedure $\mathcal{P}$. After that, the "summaries" marked as "non-ASCII", containing less than 5 tokens or only consist of punctuation are regarded as trivial "summaries".

Finally, the PRs with empty or trivial "summaries" or with less than 2 or more than 20 valid commits are removed from the dataset. The max lengths of "articles" and "summaries" are set 400 and 100, respectively, and PRs with long "articles" or long "summaries" are also removed.

As a result, a dataset with 41,932 PRs is constructed.

5.4.4 *Evaluation*

PRSummarizer was evaluated on the dataset mentioned above in terms of ROUGE. The ROUGE-1, ROUGE-2 and ROUGE-L scores of PRSummarizer are 34.15, 22.38 and 32.41, respectively, which improves over two baselines, i.e., LeadCM and LexRank, by 11.57% to 25.40%. An ablation study presented that the pointer generated and the RL loss are effective and helpful for boosting PRSummarizer effectiveness. [2] also conducted a human evaluation which showed their approach performs better than the baselines and can generate more high-quality PR descriptions. In summary, the customized seq2seq model is shown to be effective for PR description generation.

5.5 Automated Generation of App Review Responses

Mobile apps have become an integral part of people's daily life. The ratings and reviews of an app in app stores reflect its user experience and quality, and can affect potential users positively or negatively. Currently, both Apple's App Store and Google Play provide a review response system

through which developers can manually respond to user feedback. In responses, developers can explain apps' functionalities, describe the roadmap of requested features, or simply express gratitude for users' opinions. Previous studies have shown that responding to app reviews can enhance app development and improve user experience [43, 44]. However, reading and responding to a large and ever-increasing number of reviews is a heavy burden for developers, if not impossible. Therefore, more automation in user review responses is important and necessary for developers.

There exist a number of works focusing on automatic user review analysis. A popular direction is classifying app reviews and identifying complaint topics in reviews [45–48]. For example, [45] designed MARA to retrieve feature requests from reviews based on linguistic rules. [47] leveraged NLP techniques and clustering algorithms to extract feature requests and suggestions from app reviews. Identifying and extracting user sentiments from app reviews is another direction [49, 50]. For instance, [49] extracted users' sentiments to different app features from app reviews based on topic modeling techniques.

However, there is limited work aiming to automate the generation of review responses. Recently, [15] took the first step by exploring the usability of seq2seq models in the app review-response dialogue scenario. This section details this work as another case study of seq2seq-model-based SAG.

5.5.1 *Problem and Characteristics*

The dialogue generation problem is generally regarded as a seq2seq learning problem. In the app review-response dialogue scenario, the source sequence $\boldsymbol{x}$ is a user view and the target sequence $\boldsymbol{y}$ is the corresponding response Although the basic seq2seq model is effective on social conversation generation, [15] argued that it is not suitable for review response generation since the app review-response dialogue has different characteristics from social conversations:

(1) The purpose of the dialogue is to further understand users' complaints and try to solve their requests, but social conversations are mainly generated for entertainment. This concern requires the model to precisely identify the problems users complain and clearly explain the solutions or express appreciation/apologies.
(2) The sentiment behind a review should be accurately identified. A review's sentiment has a significant impact on the content of its response.

For example, the response of a positive review may express appreciation, while that of a negative review may need to apologize.

(3) Most app review-response dialogues are one-round dialogue and the reviews are usually short in length. These imply the text of a review may not provide enough information for response generation.

5.5.2 *The Seq2Seq Model: RRGen*

Considering the above-mentioned characteristics, [15] proposed a novel approach named RRGen to automatically generate responses from app reviews. The overall framework of RRGen is presented in Fig. 5.5. RRGen is grounded in the basic encoder-decoder model, but explicitly integrates two kinds of context information, i.e., the high-level attributes and the keywords of reviews, in the seq2seq model.

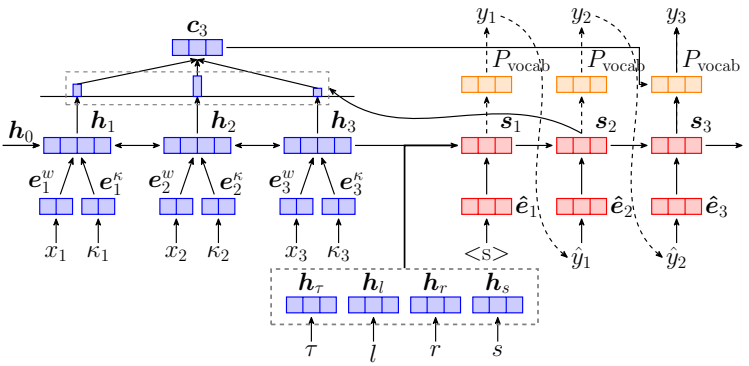


Fig. 5.5 **The framework of RRGen.** RRGen enhances the basic attentional encoder-decoder model by explicitly extracting and integrating the high-level attributes and the keywords of reviews. κ_i refers to the keyword type of x_i. $\boldsymbol{e}_i^w$ and $\boldsymbol{e}_i^\kappa$ are the embeddings of x_i and κ_i respectively, and they are concatenated before being input into the encoder's RNN. τ, l, r and s refer to app category, review length, rating and sentiment, respectively. Their representations, i.e., $\boldsymbol{h}_\tau$, $\boldsymbol{h}_l$, $\boldsymbol{h}_r$ and $\boldsymbol{h}_s$, are used to compute the decoder's initial state.

5.5.2.1 *Integrating High-Level Attributes*

RRGen considers four high-level attributes: app category, review length, rating, and sentiment. App category is app-level information. The major topics of different app categories are different and may affect the content and focuses of review responses. Review length, rating, and sentiment are

review-level information. The length of a user review reflects whether the review is informative or not. The rating and the sentiment directly affect how developers should respond to it.

Given a user review of an app, the extraction of app category, review length, and rating are straightforward. To extract its sentiment, RRGen first divides the review into sentences. Then, it calculates each sentence's sentiment score using [49]'s method. The sentiment of the review is defined as the rounded average sentiment score of all sentences in this review.

To integrate the four attributes into the seq2seq model, RRGen first converts them into their categorical forms and represents the categorical values into continuous vectors using multilayer perceptions (MLPs), i.e., a fully connected neural layer. Such continuous vectors are referred to as attribute embeddings. As an example, the embedding of app category τ is calculated as follows:

$$\boldsymbol{h}_\tau = tanh(\boldsymbol{W}_\tau Emb(\tau)), \forall \tau = 1, 2, ..., N_\tau$$

where $\boldsymbol{W}_\tau$ is the parameter matrix of the MLP, Emb is a general embedding layer. Other attribute embeddings are computed in the same way using different MLPs and embedding layers. The four attribute embeddings and the last hidden state $\boldsymbol{h}_{|\boldsymbol{x}|}$ of the encoder are concatenated as the initial hidden state of the decoder, as follows:

$$\boldsymbol{s}_0 = tanh(\boldsymbol{W}_h[\boldsymbol{h}_{|\boldsymbol{x}|} : \boldsymbol{h}_\tau : \boldsymbol{h}_l : \boldsymbol{h}_r : \boldsymbol{h}_s])$$

where $\boldsymbol{W}_h$ are learnable parameters and $\boldsymbol{h}_\tau$, $\boldsymbol{h}_l$, $\boldsymbol{h}_r$, $\boldsymbol{h}_s$ are the embeddings of app category, review length, rating and sentiment, respectively.

5.5.2.2 *Integrating Keywords*

The keywords in a user review are generally related to the topics of this review. RRGen also integrates keyword information in user reviews to help the decoder decide which word to attend to during response generation.

RRGen adopts the keyword dictionary built by [46] to identify keywords and their corresponding topics in user reviews. In detail, [46] manually summarized 12 topics that are commonly covered by user reviews and built the keyword dictionary based on WordNet [51]. For each topic, the dictionary provides a bunch of keywords related to it.

To integrate keyword information, RRGen constructs a keyword sequence $\boldsymbol{\kappa}$ based on the identified keywords and topics. The keyword token κ_i of token x_i is the topic that x_i belongs to. For example, the keyword sequence of the user review "lot of ad !" is "<O> <O> <C> <O>",

where "ad" is a keyword for topic *contents* and hence is marked by symbol "<C>". Other source tokens are non-topical words and are labeled as "<O>". Given a user review, each keyword token κ_i is first mapped into a continuous vector, i.e., an embedding, through an MLP, as follows:

$$\boldsymbol{e}_i^{\kappa} = tanh(\boldsymbol{W}_{\kappa} Emb(\kappa_i)), \forall i = 1, 2, ..., |\boldsymbol{\kappa}|$$

Next, the keyword-enhanced embedding of each input token x_i is learned through another MLP with the concatenation of its keyword embedding $\boldsymbol{e}_i^{\kappa}$ and its word embedding $\boldsymbol{e}_i^{w}$ as input, as follows:

$$\boldsymbol{e}_i = tanh(\boldsymbol{W}_e[\boldsymbol{e}_i^{\kappa} : \boldsymbol{e}_i^{w}])$$

$\boldsymbol{e}_i$ is then input into the encoder's RNN for further processing.

5.5.3 *Dataset*

To build the dataset to train and evaluate RRGen, [15] first selected the most popular 100 free apps in 2016 from Google Play. Then, they removed the apps that are not available in Google Play in April 2018 and the apps with less than 100 user reviews, and obtained 72 apps. The user reviews and responses of these apps from April 2016 to April 2018 are crawled by a Google Play crawler. In total, they collected 318,973 review-response pairs.

User reviews and their corresponding responses may contain noisy words, e.g., URLs, email addresses and misspelled words, which can affect the performance of seq2seq models. To deal with such noisy words, [15] first lowercased and lemmatized the tokens in user reviews and developer responses. Then, digits, email addresses, URLs, app names and user names were replaced with special tokens and repetitive words and misspelled words were rectified through manually summarized rules. After removing reviews with empty or one-alphabet texts, 309,246 review-response pairs are obtained to construct the dataset.

5.5.4 *Evaluation*

RRGen was first evaluated using BLEU and obtained a BLEU-4 score of 36.17. It outperforms three baselines, i.e., Random Selection, an IR-based method named *NNGen* [8], and the basic attentional encoder-decoder model by large margins. [15] also conducted an ablation study, which showed that each context information integrated into RRGen, i.e., the four high-level attributes and the keywords, is beneficial on its own, and the combination of all context information achieves the highest improvements.

Moreover, a human evaluation further confirmed the better performance of RRGen compared to the baselines. In summary, by incorporating task-specific context information into the basic seq2seq model, RRGen is shown to be effective and powerful on the automatic generation of app review responses.

5.6 Case Study Comparison

The three case studies focus on different SAG tasks, i.e., the generation of code comments, PR descriptions and app review responses. These tasks have different sets of features. To handle the task features and perform effective generation, the approach described in each case enhances and customizes the basic attentional encoder-decoder model in its own way. Table 5.1 summarizes the task features covered in the case studies and presents how the aforementioned approaches deal with them.

We can see that some features are shared among multiple tasks. For example, all three tasks require the corresponding approaches to process and fuse multiple input information. In detail, for code comment generation, Hybrid-DeepCom needs to fuse the lexical and syntactic information of code. The input of PRSummarizer contains both the commit messages and the added comments in a PR, which are combined as a compound source sequence for processing. To generate app review responses, RRGen needs to take into account user reviews, keyword information, and the high-level attributes of reviews.

Another point is that even for the same feature, different tasks may have different preferences, hence the corresponding approaches may handle the feature in different ways. As an example, to generate code comments, Hybrid-DeepCom pays more attention to the representation of OOV tokens and reduces OOV tokens by splitting identifiers. But PRSummarizer focuses more on the generation of OOV tokens and thus adopts the pointer generator. This difference mainly results from the different input of the two tasks. The input of the code comment generation task is source code, which contains plenty of developer-named identifiers, while the PR description generation task takes as input commit messages and code comments, where usually only a few identifiers may appear. So, reducing OOV tokens is not as crucial to PRSummarizer as to Hybrid-DeepCom. However, the two ways to handle OOV tokens are not competing but complementary. Thus, an approach that both splits identifiers and adopts pointer generators may achieve better performance than existing ones.

Table 5.1 How different approaches deal with different features of software artifact generation tasks.

Features	Hybrid-DeepCom	PRSummarizer	RRGen
Handling OOV tokens	Split identifiers	Pointer Generator	/
Capturing syntactic information of code	SBT AST encoder	/	/
Fusing multiple input information	Two encoders Fused attention	Compound source sequence	Keyword-enhanced source sequence Special decoder initial state
Bridging training and evaluation gaps	/	RL loss	/
Identifying sentiments	/	/	Sentiment Analysis Special decoder initial state

Also, we find that a task may have unique features. For instance, only Hybrid-DeepCom needs to capture syntactic information of code, because other approaches do not take source code as input. Another example is that sentiment analysis is only critical for generating app review responses, since source code, code comments and change documents are usually neutral.

These findings can further answer the question that why these tasks cannot be solved by the same model: because they have different input and output and thus have different feature sets. A technique or a neural component that is useful for tackling one task may be unnecessary or even harmful to address another one. Therefore, before using seq2seq models to generate software artifacts, it is important to analyze the characteristics of the specific task.

5.7 Challenges and Opportunities

The aforementioned case studies show that seq2seq models are promising for automatically generating code comments, pull request descriptions and app review responses. Based on them, we also notice the following challenges and opportunities in this direction:

5.7.1 *Challenges*

(1) **Limited Accuracy**. Because incorrect information in software artifacts may mislead developers to introduce bugs, a generative model should be quite accurate before being applied in practice. However, existing seq2seq-model-based approaches are still not powerful enough (usually, BLEU<40 and ROUGE<40) for practical applications.
(2) **Lack of uniform datasets**. For many SAG tasks, there do not exist uniform datasets that are widely acknowledged as high quality and standard. Researchers usually need to build a task-specific dataset from the beginning, which may prevent the advances of this direction.
(3) **Long code and long texts**. Existing seq2seq models and hardware, e.g., GPUs, are not powerful enough to deal with long sequences. Currently, the works in this direction always focus on modeling and generating short software artifacts. How to handle long code and long texts using seq2seq models is still an open problem.

5.7.2 *Opportunities*

(1) **Abundant and diverse open data**. With the popularity of open source culture and collaborative coding platforms (e.g., GitHub), one can freely access more and more software artifacts. Moreover, for an open source project, almost all the artifacts produced throughout its whole life cycle are publicly available. Such open data provide great opportunities for data-driven approaches. For example, it is possible to collect and build datasets of larger orders of magnitude for diverse SAG tasks, which can help to train deeper and advanced seq2seq models and bring in performance improvement. In addition, it would be interesting to recover and take advantage of connections among large-scale and different kinds of artifacts to enhance existing generative models.

(2) **Better artifact representations based on unsupervised pre-training and transfer learning**. Learning good representations of source code and/or software documents is critical for seq2seq-model-based SAG. Most existing works relied on labeled data, e.g., code-comment pairs, to learn such representations from scratch. On the one hand, the amount of labeled data for a specific task may be limited, restricting the effectiveness of trained seq2seq models. On the other hand, large unlabeled text corpora and source code corpora are abundant, and good text representations learned from unlabeled texts have shown to be useful for improving performance on many NLP tasks [52–54]. Therefore, it can be promising to learn to represent software documents or source code from large-scale unlabeled data, such as GitHub repositories, and transfer learned representations to diverse SAG tasks. As an example, BERT [54] is a breakthrough encoder which is first pre-trained on a large corpus of unlabeled text and can be effectively applied to diverse NLP tasks through supervised fine-tuning. For SAG, we can also pre-train BERT-like models on large unlabeled corpora of software documents and source code. Such pre-trained models can be used as encoders, concatenated with diverse decoders, and fine-tuned using the parallel corpus of the target SAG task. The fine-tuned encoders, or in other words, representations, can be expected to boost performance on diverse SAG tasks.

(3) **Combining rule-based, IR-based and probabilistic-model-based approaches**. The three types of approaches have been used to generate software artifacts. Rule-based methods require manual efforts to design templates and rules, but can achieve high accuracy and

may be able to deal with long texts. IR-based methods cannot handle samples that are not similar to any training sample, but work well for similar samples. Probabilistic-model-based methods, such as seq2seq models, are not accurate enough for now and can hardly handle long sequences, but they are capable of learning patterns from data and are more general. In the future, combining different kinds of methods would be a promising direction. For example, to generate commit messages by combining rule-based and probabilistic-model-based approaches, we can first manually summarize the common templates of commit messages, such as "This commit is for A, adds B, deletes C and modifies D". Probabilistic models are then leveraged to choose the most proper template and replace each placeholder in such template by generating the corresponding content from the commit, e.g., generating phases to replace "A", "B", "C" and "D" in the aforementioned template.

5.8 Summary

Software artifacts, such as code comments and commit messages, are valuable for program comprehension and software maintenance. However, developers sometimes neglect writing or maintaining them. This problem can be alleviated by automated software artifact generation. This chapter introduces the research direction of sequence-to-sequence-model-based automated software artifact generation. We describe the preliminary knowledge of sequence-to-sequence learning, demonstrate how to adopt and customize sequence-to-sequence models to tackle specific software artifact generation tasks through three case studies, summarize and discuss the features of the tasks handled in these case studies, and point out several challenges and opportunities of this research area. Hopefully, this chapter can inspire researchers and developers to learn generation patterns for more software artifacts using sequence-to-sequence models, to propose more powerful and accurate sequence-to-sequence models for software artifact generation, and to apply these generative models in practice to help developers write software artifacts and understand software.

References

[1] R. Dyer, H. A. Nguyen, H. Rajan and T. N. Nguyen, Boa: A language and infrastructure for analyzing ultra-large-scale software repositories, in

Proceedings of the International Conference on Software Engineering. IEEE, pp. 422–431 (2013).

[2] Z. Liu, X. Xia, C. Treude, D. Lo and S. Li, Automatic generation of pull request descriptions, in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, pp. 176–188 (2019).

[3] R. P. Buse and W. R. Weimer, Automatically documenting program changes, in *Proceedings of the International Conference on Automated software engineering*, pp. 33–42 (2010).

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock and K. Vijay-Shanker, Towards automatically generating summary comments for java methods, in *Proceedings of the International Conference on Automated software engineering*, pp. 43–52 (2010).

[5] M. Linares-Vásquez, B. Li, C. Vendome and D. Poshyvanyk, Documenting database usages and schema constraints in database-centric applications, in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 270–281 (2016).

[6] S. Haiduc, J. Aponte, L. Moreno and A. Marcus, On the use of automated text summarization techniques for summarizing source code, in *Proceedings of the Working Conference on Reverse Engineering*. IEEE, pp. 35–44 (2010).

[7] E. Wong, T. Liu and L. Tan, Clocom: Mining existing source code for automatic comment generation, in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, pp. 380–389 (2015).

[8] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing and X. Wang, Neural-machine-translation-based commit message generation: how far are we? in *Proceedings of the International Conference on Automated Software Engineering*, pp. 373–384 (2018).

[9] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation, in *Proceedings of the Conference on Program Comprehension*, pp. 200–210 (2018).

[10] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, Deep code comment generation with hybrid lexical and syntactical information, *Empirical Software Engineering*, pp. 1–39 (2019).

[11] A. LeClair, S. Jiang and C. McMillan, A neural model for generating natural language summaries of program subroutines, in *Proceedings of the International Conference on Software Engineering*. IEEE, pp. 795–806 (2019).

[12] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu and P. S. Yu, Improving automatic source code summarization via deep reinforcement learning, in *Proceedings of the International Conference on Automated Software Engineering*, pp. 397–407 (2018).

[13] S. Jiang, A. Armaly and C. McMillan, Automatically generating commit messages from diffs using neural machine translation, in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, pp. 135–146 (2017).

[14] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du and Y. Qian, Generating commit messages from diffs using pointer-generator network, in *Proceedings of the*

International Conference on Mining Software Repositories. IEEE, pp. 299–309 (2019).

[15] C. Gao, J. Zeng, X. Xia, D. Lo, M. R. Lyu and I. King, Automating app review response generation, in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, pp. 163–175 (2019).

[16] D. E. Rumelhart, G. E. Hinton and R. J. Williams, Learning representations by back-propagating errors, *Nature* **323**, 6088, pp. 533–536 (1986).

[17] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*. MIT press (2016).

[18] S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural computation* **9**, 8, pp. 1735–1780 (1997).

[19] K. Cho, B. van Merriënboer, D. Bahdanau and Y. Bengio, On the properties of neural machine translation: Encoder–decoder approaches, in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111 (2014).

[20] M. Schuster and K. K. Paliwal, Bidirectional recurrent neural networks, *IEEE transactions on Signal Processing* **45**, 11, pp. 2673–2681 (1997).

[21] D. Bahdanau, K. Cho and Y. Bengio, Neural machine translation by jointly learning to align and translate, in *Proceedings of the International Conference on Learning Representations* (2015), `http://arxiv.org/abs/1409.0473`.

[22] X. Gu, H. Zhang, D. Zhang and S. Kim, Deep api learning, in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 631–642 (2016).

[23] K. Papineni, S. Roukos, T. Ward and W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in *Proceedings of the Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, pp. 311–318 (2002).

[24] S. Banerjee and A. Lavie, Meteor: An automatic metric for mt evaluation with improved correlation with human judgments, in *Proceedings of the Acl Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pp. 65–72 (2005).

[25] M. Denkowski and A. Lavie, Meteor universal: Language specific translation evaluation for any target language, in *Proceedings of the Workshop on Statistical Machine Translation*, pp. 376–380 (2014).

[26] C.-Y. Lin, ROUGE: A package for automatic evaluation of summaries, in *Text Summarization Branches Out*. Association for Computational Linguistics, pp. 74–81 (2004), `https://www.aclweb.org/anthology/W04-1013`.

[27] S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, Summarizing source code using a neural attention model, in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083 (2016).

[28] X. Hu, G. Li, X. Xia, D. Lo, S. Lu and Z. Jin, Summarizing source code with transferred api knowledge, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 19, pp. 2269–2275 (2018).

[29] G. Sridhara, L. Pollock and K. Vijay-Shanker, Automatically detecting and describing high level actions within methods, in *Proceedings of the International Conference on Software Engineering*. IEEE, pp. 101–110 (2011).

[30] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock and K. Vijay-Shanker, Automatic generation of natural language summaries for java classes, in *Proceedings of the International Conference on Program Comprehension*. IEEE, pp. 23–32 (2013).

[31] P. W. McBurney and C. McMillan, Automatic documentation generation via source code summarization of method context, in *Proceedings of the International Conference on Program Comprehension*, pp. 279–290 (2014).

[32] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch and S. D'Mello, Improving automated source code summarization via an eye-tracking study of programmers, in *Proceedings of the International Conference on Software engineering*, pp. 390–401 (2014).

[33] M. Allamanis, H. Peng and C. Sutton, A convolutional attention network for extreme summarization of source code, in *Proceedings of the International Conference on Machine Learning*, pp. 2091–2100 (2016).

[34] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, A survey of machine learning for big code and naturalness, *ACM Computing Surveys* **51**, 4, pp. 1–37 (2018).

[35] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus and G. Canfora, Automatic generation of release notes, in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 484–495 (2014).

[36] S. Rastkar and G. C. Murphy, Why did this code change? in *Proceedings of the International Conference on Software Engineering*. IEEE, pp. 1193–1196 (2013).

[37] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte and D. Poshyvanyk, On automatically generating commit messages via summarization of source code changes, in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. IEEE, pp. 275–284 (2014).

[38] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte and D. Poshyvanyk, Changescribe: A tool for automatically generating commit messages, in *Proceedings of the International Conference on Software Engineering*, Vol. 2. IEEE, pp. 709–712 (2015).

[39] J. Shen, X. Sun, B. Li, H. Yang and J. Hu, On automatic summarization of what and why information in source code changes, in *Proceedings of the Annual Computer Software and Applications Conference*, Vol. 1. IEEE, pp. 103–112 (2016).

[40] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus and G. Canfora, Arena: an approach for the automated generation of release notes, *IEEE Transactions on Software Engineering* **43**, 2, pp. 106–127 (2016).

[41] A. See, P. J. Liu and C. D. Manning, Get to the point: Summarization with pointer-generator networks, in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083 (2017).

[42] S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross and V. Goel, Self-critical sequence training for image captioning, in *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pp. 7008–7024 (2017).

[43] S. McIlroy, W. Shang, N. Ali and A. E. Hassan, Is it worth responding to reviews? studying the top free apps in google play, *IEEE Software* **34**, 3, pp. 64–71 (2015).

[44] S. Hassan, C. Tantithamthavorn, C.-P. Bezemer and A. E. Hassan, Studying the dialogue between users and developers of free apps in the google play store, *Empirical Software Engineering* **23**, 3, pp. 1275–1312 (2018).

[45] C. Iacob and R. Harrison, Retrieving and analyzing mobile apps feature requests from online reviews, in *Proceedings of the Working Conference on Mining Software Repositories*. IEEE, pp. 41–44 (2013).

[46] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora and H. C. Gall, What would users change in my app? summarizing app reviews for recommending software changes, in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 499–510 (2016).

[47] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci and A. De Lucia, Recommending and localizing change requests for mobile apps based on user reviews, in *Proceedings of the International Conference on Software Engineering*. IEEE, pp. 106–117 (2017).

[48] C. Gao, W. Zheng, Y. Deng, D. Lo, J. Zeng, M. R. Lyu and I. King, Emerging app issue identification from user feedback: experience on wechat, in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE, pp. 279–288 (2019).

[49] E. Guzman and W. Maalej, How do users like this feature? a fine grained sentiment analysis of app reviews, in *Proceedings of the International Requirements Engineering Conference*. IEEE, pp. 153–162 (2014).

[50] X. Gu and S. Kim, "What parts of your apps are loved by users?"(t), in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, pp. 760–770 (2015).

[51] G. A. Miller, Wordnet: a lexical database for english, *Communications of the ACM* **38**, 11, pp. 39–41 (1995).

[52] Y. Qi, D. Sachan, M. Felix, S. Padmanabhan and G. Neubig, When and why are pre-trained word embeddings useful for neural machine translation? in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 529–535 (2018).

[53] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa, Natural language processing (almost) from scratch, *Journal of Machine Learning Research* **12**, ARTICLE, pp. 2493–2537 (2011).

[54] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186 (2019).

https://doi.org/10.1142/9789811239922_0006

Chapter 6

Machine Learning to Support Code Reviews in Continuous Integration

Miroslaw Staron[a], Miroslaw Ochodek[b], Wilhelm Meding[c],
Ola Söder[d] and Emil Rosenberg[e]
[a] *Chalmers University of Gothenburg*
[b] *Poznan University of Technology*
[c] *Ericsson AB*
[d] *Axis Communications*
[e] *Saab AB*

6.1 Introduction

The paradigm shift from simple Agile practices to Continuous Integration (CI) brought a different dynamics into software development [1, 2]. Five to ten years ago, software companies focused heavily on CI, initiating, and monitoring, activities aimed at getting the CI machinery working. Activities involved e.g. efficient use of new tools, (e.g. SonarQube, Gerrit, Kubernetes), high focus on code reviews, and change of ways-of-working to adapt to the efficient set-up and working of the CI machinery. Agile teams, together with the CI team, program managers, releases managers, tools team and more, all coordinated their effort to enable their organization to become a CI efficient organization.

The last few years the focus has widened to include CDs (i.e. continuous deliveries and continuous deployments). At the same time, (former) new technologies as cloud, 5G, containerization, microservices and more has put

a whole new attention and focus on CI, since CI is the very prerequisite for CI/CD. For CI, being efficient and having good quality, is far from enough. Reality today demands that the main branch holds such quality, that the main branch is always open for deliveries from the agile teams and it is always available for deliveries to customers. This puts a lot of pressure on the organization using the CI flow. For instance, testing must be automated and cover all possible aspects [3], and the code reviews have to be frequent and effort-consuming.

Keeping high quality of the code base entails the use of a number of quality assurance steps, between the check-ins of the code by the designers and its integration into the main branch [4]. These steps include static analysis of the code (e.g. using Lint, [5]) and manual code reviews [6]. Despite their obvious benefits, these methods have several downsides. Manual reviews are time consuming, effort intensive and often reviewer-dependent. Even Linus Torvalds, the creator of Linux, identifies the need for coding review as one of the crucial activities, at the same time acknowledging that it is a time and effort consuming activity [7]. Therefore, automation of this process is called for by the software engineering industry in general, and by our industrial case companies in particular.

In our previous work (Ochodek *et al.* [8]) we studied how to find arbitrary, company-specific, coding violations and using examples to train machine learning classifiers to find similar code fragments. The scenario was to find examples of good and smelly code, use these examples to train a machine learning classifier and look for more examples of the smelly code. The limitation, however, was the need to create a sufficient number of examples for each type of smell in the code and to maintain the examples as the organization evolved.

Therefore, in this chapter, we present a method that addresses this limitation. Instead of manually creating the examples and label the code fragments based on whether the example shows good code or a code smell, we use review comments from code review tools to label the code. In particular, we address the research problem of how to automatically process the large number of code patches submitted in CI flows, extract the rules for coding guidelines automatically and recommend code fragments/lines for manual reviews from the perspective of software designers. Mostly, these are code reviews done by experienced designers and reviewers in tools like Gerrit [9].

This chapter is based on our experiences from a number of research projects, conducted according to the action research methodology [10].

During the project we had the unique opportunity to work with two organizations, where we studied their code reviews and suggested changes. The chapter uses the same techniques and methods, but is based on the data from open source repositories to provide the readers with the opportunity to replicate the study and to reuse the tools and the data.

The main contribution of the chapter is the hands-on, step-by-step presentation of the method and the demonstration of its potential. It shows how the modularity of the method can be utilized to develop even more complex code analysis tools and how to use the method in reader's own context.

The remaining of the chapter is structured as follows. Firstly, Sec. 6.2 presents how code review processes work in modern CI toolchains. Section 6.3 describes the workflow for automated review analysis and recommendation. Sections 6.4, 6.5, 6.6 and 6.7 present the details of our approach. Section 6.9 presents a full example from the open source wireshark protocol implementation. Section 6.10 discusses the customization of the workflow with more classifiers, different source systems and different feature extraction techniques. Section 6.11 presents further reading in this area, for the readers who would like to get more in-depth understanding of the techniques and methods presented in the study. Finally Sec. 6.12 presents the conclusions and further work from our study.

6.2 Code review in CI

Modern code reviews have evolved from being a physical meeting between the reviewer and the author to a collaborative activity supported by dedicated tools [11]. Figure 6.1 presents a typical code review workflow in a continuous integration context. The grey background in the figure shows which activities were in the scope of the automated code review support described in this chapter.

Step 1: Software designer clones the repository, thus copies the original code base to own workstation.

Step 2: Once done with adding new feature/bug fix, the designer commits the code to the integration.

Step 3: Gerrit executes project-specific static analysis checks and executes tests according to the test plan.

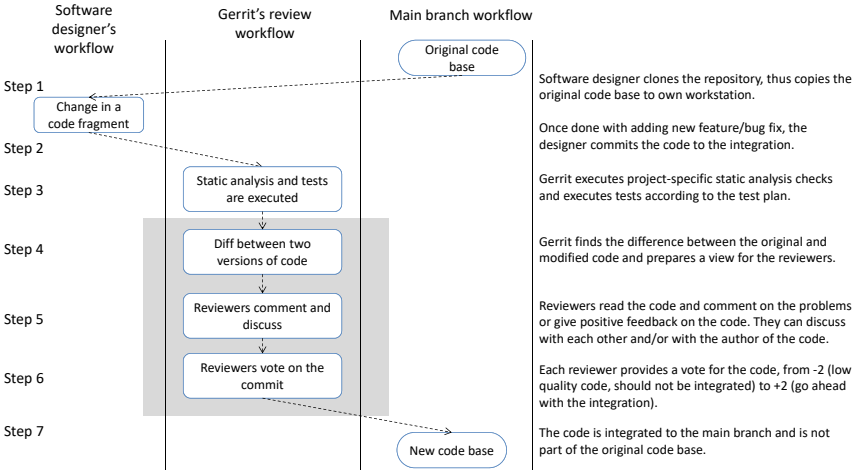


Fig. 6.1: A typical code review workflow in continuous integration projects.

Step 4: Gerrit finds the difference between the original and modified code and prepares a view for the reviewers. The standard set-up of the tools identifies all changes (added, removed and modified code) and presents that the to reviewers. The modifications vary from small (a few lines modified in a single file) to quite extensive (multiple code fragments added, removed and modified in multiple files).

Step 5: Reviewers read the code and comment on the problems or give positive feedback on the code. They can discuss with each other and/or with the author of the code. The process of reading the code, if done properly, requires the designers to read the commit messages to understand what was done (e.g. which feature was implemented or which defect was fixed). The extensive code commits are thus effort intensive and time consuming, which, together with scarce documentation, can lead to long review durations. Pinpointing "suspicious" code fragments could reduce the effort required and thus the duration.

Step 6: Each reviewer provides a vote for the code, from -2 (low quality code, should not be integrated) to $+2$ (go ahead with the integration). The voting is done for the entire commit, but provides the basic view on the code quality — good quality code is up-voted and low quality code is down-voted. As the voting is mandatory for all commits, this presents the

opportunity to analyze and understand what good and bad quality means in terms of code constructs.

Step 7: The code is integrated to the main branch and is not part of the original code base.

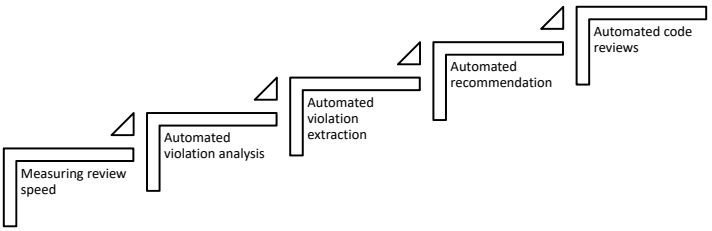
Steps 3–6 are often repeated several times until the reviewers are satisfied with the changes. In practice this can mean that these steps can take over four iterations. So, even small improvements in that loop can bring significant savings. Our focus on the three greyed steps is also dictated by the fact that these activities are dependent on human reviewers and therefore can be affected by external factors. For example, the reviewer can be unavailable due to his/her commitments to other projects, or the reviewers may not fully agree on the proposed change. These factors can play a significant role when the number of code commits is large and therefore the effort required from the reviewers is high. Reducing the number of manual reviews would help to optimize the process and therefore lead to the improvement of the overall speed and quality of software development [12].

Our industrial partners identified the following challenges which need to be addressed in this process:

(1) Human involvement from the beginning — not all commits require manual review, but involving human reviewers leads to, paradoxically, lower review quality; human reviewers often miss important code fragments in the constant inflow of the patches. Not enough focus on things that really matters, things that don't go away as soon as the compiler has done its job. Filtering out the commits that do not require manual review would have a positive effect, both on product quality and the spreading of knowledge from senior to junior developers.
(2) Frustration in the iterative process — since steps 3–6 can be repeated several times, code authors and code reviewers can discuss for a long time with long breaks between each discussion comment, which slows down feature development and leads to frustration in the team.
(3) Company specific coding rules are costly to maintain — the set of rules might be huge, they might change all the time. Even worse, some of the languages used in large scale, highly specialized software organizations, might be domain specific, or consist of an unholy mix of established languages that of the shelf tools can't handle.
(4) The review process that's used for code is often used for other types of machine readable "text" as well — configuration files and so on. This

is also something that's hard to manage with conventional of the shelf tools.

However, the full automation of the review process is not desired. The process of reviewing code in CI is also a process of learning — knowledge from the experienced designers is transferred to the junior ones. The code and design decisions are discussed and therefore improved, or at least the tradeoffs are taken responsibly and after impact analyses. Therefore, we use the following metaphor of the review automation stairway in our work, Fig. 6.2. The staiway is inspired by the organizational performance stairway [13] and symbolizes how an organization can elevate its competence in code reviewing without jeopardizing the organizational learning process.



Review speed measurement	Automated	Automated	Automated	Automated	Automated
Review data extraction	Manual	Automated	Automated	Automated	Automated
Rule definition	Manual	Manual	Assisted	Automated	Automated
Code analysis	Manual	Manual	Assisted	Automated	Automated
Feedback	Manual	Manual	Manual	Manual	Automated

Fig. 6.2: Stairway in Automation of Code Review Processes in studied companies.

The first stage of is **measuring review speed**, where the organization automates the measurement of speed of the review process [12]. The other activities are manual:

- Review data extraction: exporting the review comments, their metadata (e.g. timestamps) and the reviewed code fragments from the code review system to a database or a file which can be used in statistical analyses (e.g. using R).
- Rule definition: defining which reviews should be considered as positive and negative, which review comments should be discarded (e.g.

unambiguous) and which review comments should be considered as coding guidelines.
- Code analysis: analyzing the source code of new commits and providing the results to the code authors.
- Feedback: providing the code reviewers with proposal for the comments for a given code fragment.

The second stage of **automated violation analysis** is when the organization starts to codify their specific coding guidelines into automated tools and use these tools to analyze the code. For example, when organizations write their own static analysis rules or style checkers. The organization automatically analyze the code fragments, but the process of defining the rules and code analyses are still manual [12].

In the third stage of **automated violation extraction**, the rule definition and code analysis is assisted. This means that the automated code review tool provides automated suggestions which review comments are repetitive and the designers can write a static analysis rule to be automatically checked. The tool can also provide examples of code fragments to which these review comments belong.

Subsequently, in the stage of **automated recommendations**, the rule definition and the code analyses are automated. This means that the system can analyze the code and provide the code authors which an annotation whether a specific code fragment violates any rules or not.

Finally, in the stage of **automated code reviews**, the system can also provide the insight which code review comments are most often used when commenting similar code fragments. Since this is the most interesting, and the most beneficial, approach, we focus on the automated code reviews in this paper.

6.3 Code analysis toolchain

Extracting code reviews and the code linked to these comments is a process which is organized into three parts:

(1) Exporting raw data from the source system.
(2) Extracting features from code and comments.
(3) Classification and recommendation.

These parts are depicted in Fig. 6.3. The flow in the figure starts with the raw data export from the Gerrit review system, which is done using

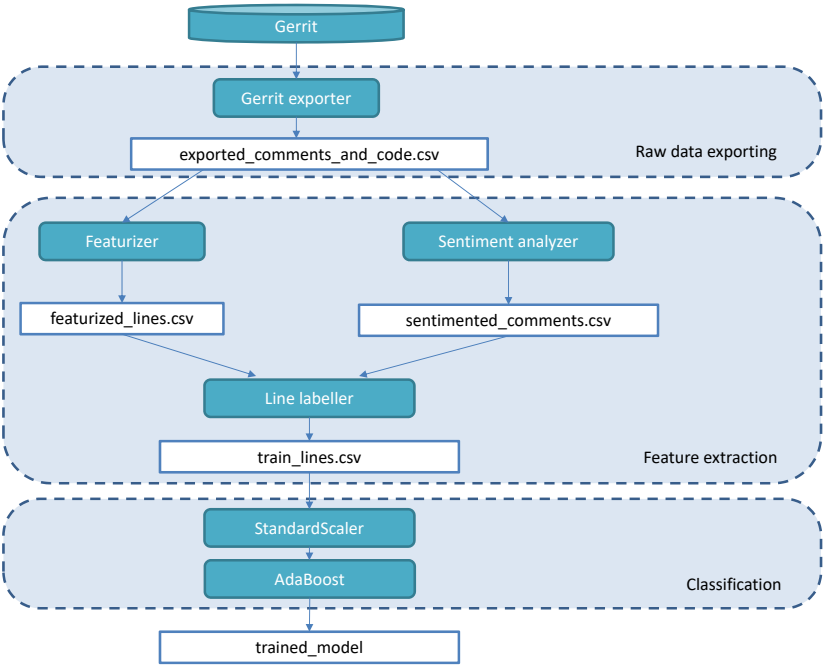


Fig. 6.3: Code analysis flow — training the classifier.

Python scripts and the JSON API to the system. Then, the flow continues to the feature extraction, which is based on the bag-of-words algorithm and finally it ends with the training of the classifier — the result is the trained ML model, which we can apply to recognize violations on new code fragments.

Figure 6.4 presents how the trained model is applied on the new code base. The flow is similar to the analysis, except that there is no sentiment analysis (as there are not comments on the new code yet) and the classifier have the new input — the trained model.

One observation to take from these diagrams is the change of complexity — in the training flow, the complexity is mostly around the concept of feature extraction and labelling of lines. In the recognition of the new code violation, the complexity is shifted toward the classification part — the classifier needs to take the trained model as the input.

In the figures we use one example of a data source — Gerrit code review system — which is one of the most popular tools. We can exchange this

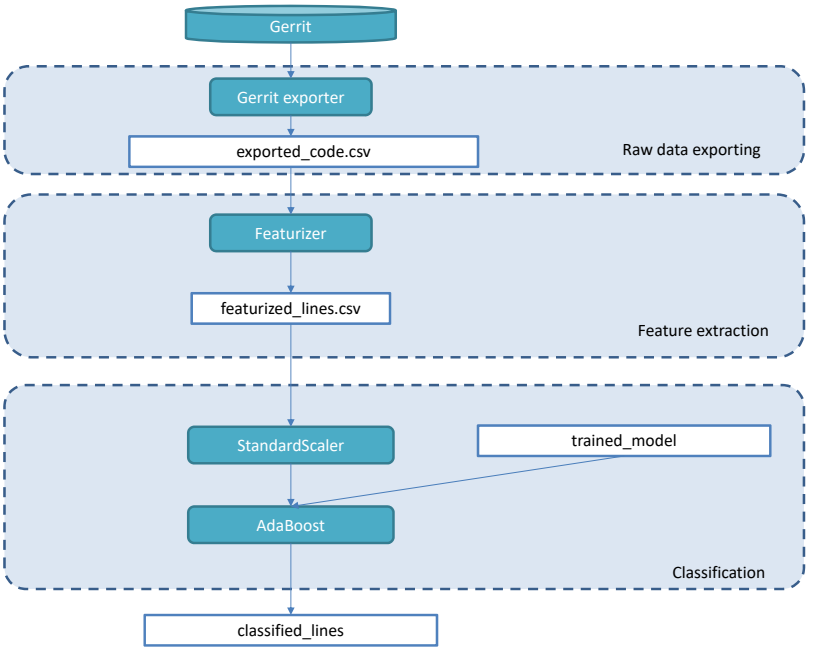


Fig. 6.4: Code analysis flow — recognizing violations on new code.

tool for others (e.g. GitLab, GitHub, Visual Studio Team System) leaving the other parts intact. The classifiers used in the figure — AdaBoost — are also an example, and can be exchanged to neural networks or other types of classifiers.

6.4 Code extraction

Before we move to the description of the algorithm, let us look at an example of how a code review looks like in a typical code review tool — Fig. 6.5. The example shows a comment related to the code in a patch that is committed to the main branch. The figure is drawn manually to emphasize the link between the comment and the code, and to abstract away the cluttering details of a code review tool. However, it is based on how Gerrit and Git present the review comments.

The figure illustrates an important design consideration — which lines are labelled as “good” and which are labelled as “bad”. In our studies, we

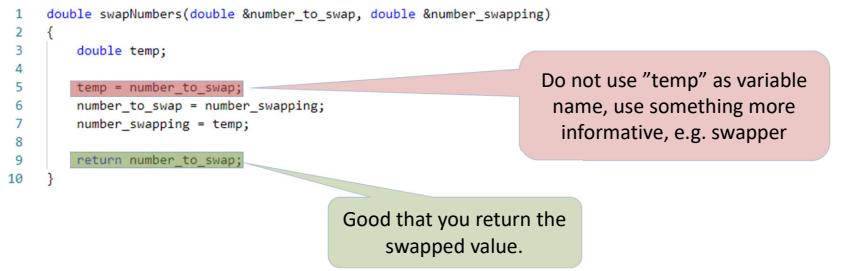


Fig. 6.5: An example of a review comment.

found that we need to export all comments and label only the lines that are commented. We experimented with exporting all lines and labelling the lines that were not commented as "good", but this is not accurate as:

(1) Reviewers unwillingly repeat their comments, instead the write comments like this "You use too many temp variables, I will not comment on every one instance, please fix it throughout the code."
(2) In large commits, the reviewers often focus on "sensitive" code fragments and tend to comment on them. The rest of the lines is not commented, but this does not mean they are correct or proper, it could just mean that the reviewer was pressed on time.

The script that extracts the lines and their comments uses the API of the code review tool. In our case, we use Gerrit, as it is a tool that is both popular and has a straightforward JSON API. Code in Fig. 6.6 presents a JSON API call to get the IDs of submitted and reviewed patches.

```
# getting the handle for the changes in a Gerrit instance
# the variable 'changes' stores the JSON string with the changes
changes = rest.get("/changes/?q=status:merged&o=ALL_FILES&o=ALL_REVISIONS&o=DETAILED_LABELS",
                   headers={'Content-Type': 'application/json'})
```

Fig. 6.6: JSON API call to retrieve a batch of patch information.

The code returns a JSON string which we can process as a collection in the subsequent part of the script — processing each patch and extracting the comments. The code is presented in Fig. 6.7.[1]

[1]For the sake of the simplicity of the example, we omit error handling, e.g. timeouts and missing elements of the JSON string.

```
1 # processing each code patch in the extracted JSON
2 for iIndex, change in enumerate(changes, start=1):
3     changeID = change['id']
4     revisions = change['revisions']
5 
6     for revID in list(revisions.keys()):
7         # JSON API call to get all comments for one revision in one patch
8         currentComment = rest.get("/changes/{}/revisions/{}/comments".format(changeID,revID),
9                                   headers={'Content-Type': 'application/json'})
10 
11        # not all revisions have comments, so we only look for those that have them
12        if len(currentComment) > 0:
13            for oneFile, oneComment in currentComment.items():
14                try:
15                    # this code extracts information about the comment
16                    # things like which file and which lines
17                    for oneCommentItem in oneComment:
18                        # if there is a specific line and characters as comments
19                        if 'range' in oneCommentItem:
20                            strStartLine = oneCommentItem['range']['start_line']
21                            strStartChar = oneCommentItem['range']['start_character']
22                            strEndLine = oneCommentItem['range']['end_line']
23                            strEndChar = oneCommentItem['range']['end_character']
24 
25                        # if the commented/reviewed line is not empty
26                        # then here is where we do it
27                        if strLine != '':
28                            # we need the line below to properly encode the filename as URL
29                            urlFileID = urllib.parse.quote_plus(oneFile)
30                            fileContentString = f'/changes/{changeID}/revisions/{revID}/files/{urlFileID}/content'
31                            fileContent = rest.get(fileContentString, headers={'Content-Type': 'application/json'})
32                            fileLines = fileContent.split("\n")
33 
34                            # if we have the lines delimitations (comment that is linked to lines)
35                            if strStartLine != '0':
36                                iStartLine = int(strStartLine) - 1
37                                iEndLine = int(strEndLine) - 1
38 
39                                for oneLine in fileLines[iStartLine:iEndLine]:
40                                    strToCSV = str(changeID) + ";" + \
41                                    str(revID) + ";" + \
42                                    oneFile + ";" + \
43                                    str(strLine) + ";" + \
44                                    str(strStartLine) + ";" + \
45                                    str(strEndLine) + ";" + \
46                                    fileHandle.write(strToCSV + "\n")
```

Fig. 6.7: Python code to process each patch — extract lines and comments.

Lines 8–9 call the Gerrit API and retrieve a comment for each revision in the current patch. The look starting in line 13 processes each comment and each file. Each comment has a start and end line, referenced as numbers, which we extract in lines 20–23. Each comment has a reference to the file which is commented, which we extract in line 29. We use the JSON API again to extract the content of the commented file in lines 30–31. In lines 35–37, we extract the source code lines to which the review comment belongs. In lines 39–46, we save the line and the comment to the .csv file for further processing — feature extraction.

6.5 Feature extraction

The feature extraction step uses two algorithms — bag-of-words for the source code analysis and sentiment analysis for the comments. The bag-of-words extraction of features is based on the work we used in our previous studies [3, 8, 14]. For the analysis of comments we use a lexicon-based sentiment analysis [15].

6.5.1 *Bag-of-words analysis of source code*

The bag-of-words model (BoW) is a simplified representation of text frequently used in machine learning. It extracts features by counting occurrences of tokens or sequences of tokens (n-grams) in the passage of text (in our case, a line of code). The method requires a vocabulary that can be automatically derived from the text or explicitly provided by the user (we can also mix these two approaches).

In Fig. 6.8, we present a snippet of code in Python that can be to extract BoW features from lines of code. In lines 5–12, we define a custom tokenizer that split lines of code using characters that have special meaning in many programming languages. In contrast to the typical implementations of BoW used for natural language processing, these splitting characters are preserved and included in the vocabulary. In lines 14–18, we use the CountVectorized class available in the sklearn library to train the BoW model. The class offers multiple parameters that can be set to control how the model is created. In the provided example, we use automatic vocabulary creation and limit the size of vocabulary to 100 most frequently appearing features. We can also control the size of n-grams extracted from the text. W show how changing the ngram_range parameter affects the extracted features. Finally, in the lines 20–22 of the snippet, we show how to prepare a two-dimensional array storing the extracted features.

In our previous studies [3, 8, 14], we proposed some extensions to the standard BoW model that help applying it to code analysis. For instance, we proposed to convert tokens being out of the vocabulary to so-called token signatures. Such signatures are created by firstly replacing each uppercase letter with "A", each lowercase letter with "a", and each digit with "0" and then shrunking each subsequence of the same characters into a single character only (e.g., *aaa* to *a* or __ to _). Finally, the process is repeated for pairs and triples of characters (e.g., *AaAa* is converted to *Aa*). Using token signatures can help reducing the size of vocabulary and preventing

```
from sklearn.feature_extraction.text import CountVectorizer
import re
import pandas as pd

CODE_STOP_DELIM = "([\s\t\(\)\[\]{}!@#$%^&*\/\+\-=;:\\\\|`'\"~,.<>/?\n])"

def code_stop_words_tokenizer(line):
    global CODE_STOP_DELIM
    split_line = re.split(CODE_STOP_DELIM, line)
    split_line = list(filter(lambda a: a != '', split_line))
    split_line = ["0" if x.isdigit() else x for x in split_line]
    return split_line

count_vect = CountVectorizer(max_features=100,
                             tokenizer=code_stop_words_tokenizer,
                             ngram_range=(1,1))

bag_of_words = count_vect.fit_transform(lines).todense()

colnames = [x for x in sorted(count_vect.vocabulary_.keys())]

lines_bow = pd.DataFrame(bag_of_words, columns=colnames)

```

Fig. 6.8: Using the bag-of-words model to extract features from code.

from overfitting models to specific names of variables or methods in the training code.

6.5.2 *Sentiment analysis of review comments*

The first part of the sentiment analysis is the configuration of the algorithm. The configuration is a list of positive and negative sentiments, as shown in Fig. 6.9.

```
# lexicon for positive sentiments
keywordsComments_positive = ['good', 'idea', 'good idea',
                             'done', 'beside', 'improved',
                             'thank' 'yes', 'well',
                             'nice', 'positive', 'better',
                             'best', 'super', 'great',
                             'fantastic']

# lexicon for negative sentiments
keywordsComments_negative = ['not good', 'not improve', "don't"
                             'should', 'should not', '?',
                             'aside', 'tend', 'not done',
                             'bad', 'improve', 'remove',
                             'add', 'include', 'not include',
                             'defeat', 'no', 'do not',
                             'chaotic', 'negative', 'worse',
                             'worst']
```

Fig. 6.9: Definition of the lexicon for the analysis of code review comments.

In our case, we used a simple lexicon, which we derived by reading ca. 200 code review comments. The analysis of every review comment is presented in Fig. 6.10. Lines 10–11 and 13–14 are the main loops that calculate the number of words from the lexicon for positive and negative sentiments. Lines 16–17 calculate the quotinent, which is necessary as the number of lexicon words can differ between the positive and negative sentiments. Line 19 calculates the sentiment and lines 24–27 recalculate it to a class for the machine learning algorithm in the next step.

```
1 def comment2sentiment(strComment,
2                       keywordsComments_positive,
3                       keywordsComments_negative):
4     countPositive = 0
5     countNegative = 0
6
7     totalPositives = len(keywordsComments_positive)
8     totalNegatives = len(keywordsComments_negative)
9
10    for oneKeyword in keywordsComments_positive:
11        countPositive += strComment.lower().count(oneKeyword.lower())
12
13    for oneKeyword in keywordsComments_negative:
14        countNegative += strComment.lower().count(oneKeyword.lower())
15
16    quotinentPositive = countPositive / totalPositives
17    quotinentNegative = countNegative / totalNegatives
18
19    sentimentQuotinent = quotinentPositive - quotinentNegative
20
21    # once we have the quotinent, we change it into verdict
22    # anything that is positive becomes 1 and
23    # anything that is negative becomes 0
24    if sentimentQuotinent > 0:
25        return 1
26    else:
27        return 0
```

Fig. 6.10: Function to analyze the sentiment and turn this into a "class" that is used for the machine learning algorithms in the next step.

We chose this method for sentiment analysis due to its simplicity and our ability to control what is considered as positive and negative comment. By re-configuring the lexicon we can calibrate the method based on our needs. For example, we can calibrate the lexicon for the specific vocabulary used by the project team or by the entire company.

The result of the sentiment analysis provides us with the classification of each comment — positive (`1`) or negative (`0`). This is used in the labelling of the lines that are commented. Each line is labelled as `1` when the comment is positive and `0` when the comment is negative. In the next step, we train a machine learning classifier to recognize lines belonging to these classes.

More precisely, we train the classifier to assign a label to each line — `1` or `2`.

6.6 Model development

We use supervised learning classifiers to develop the model as we have the labelled data to train the model. In our studies, we experimented with two types of models — whitebox models based on decision trees and blackbox models based on convolutional neural networks. The innerworkings of the models are significantly different, but for the purpose of classifying lines, they are interchangeable, i.e. we can plug-in and out different models into the same workflow.

The features extracted from the source code are often many (over 1,000 features) as they need to capture the variability of the source code, but the classes are only two. This means that the classifier needs to be robust to multidimensional data and needs to be able to capture the variability. We use the AdaBoost classifier with multiple decision trees as the classifier which provides a good trade-off between the time needed for training, the data required and the resulting classification performance.

Figure 6.11 presents the code for the training of the classifier with its 10-fold cross-validation.

```
1 ab_pipeline = Pipeline([
2         ('std_scaler', StandardScaler()),
3         ('ab', AdaBoostClassifier(DecisionTreeClassifier(max_depth=20),
4                                   n_estimators=600,
5                                   algorithm="SAMME.R",
6                                   learning_rate=0.2)),
7     ])
8
9 cross_val_score(ab_pipeline, X, y, cv=10, scoring='f1')
```

Fig. 6.11: Code to train the AdaBoost classifier based on CART decision trees.

Lines 1–7 define the classifier, which is organized into a pipeline. The first step is to use the `StandardScaler`, which is a pre-processing algorithm that scales features by removing the mean and scaling to unit variance. The second step is the AdaBoost algorithm, which is based on the `DecisionTreeClassifier`. The `DecisionTreeClassifier` is a Python implementation of the CART decision tree algorithm [16]. The algorithm is well-known and robust, providing a good platform for the building the AdaBoost ensemble classifier [17].

The important parameter is the number of decision trees in the classifier, which is line 4. In this case, we have 600 trees. This is a large number and it depends on the number of features in the data set. For this code, we ended-up with over 2,000 features, which justified such a large number of trees. Based on our experiments, we recommend to set the parameter to be ca. 30% of the number of features.

Line 9 is the code that executes the training and validation process. It randomly splits the data set into train and test parts, then training the classifier 10 times (`cv` parameter). The result is the trained model, which is stored in the `ab_pipeline` variable.

The trained classifier is ready to help us to make predictions whether a specific line should be reviewed manually or not.

6.7 Making a recommendation

Making the recommendation is based on the application of the trained classifier, which is programmed using a single line of code — `y_pred_ab = ab_pipeline.predict(X_test)`. In this line, we provide the input to the classifier, `X_test`, which is a feature vector of one line. The result is the predicted class, `y_pred_ab`, which is either a `0` for the line that violates the programming practice or `1` for the line which does not.

Since our classifier was trained based on the classes obtained from the sentiment analysis, this predicted class `0` means, essentially, that "if this line was reviewed by a reviewer, the reviewer would react negatively to this line". Having this knowledge we can continue building systems on top of this, e.g. looking for similar lines in the training set, identifying their comments and providing the recommendation to the reviewers based on this identified comment.

In this workflow, however, we do not work with the semantics of the comments and therefore we should be careful with the automated recommendations. The comments can irrelevant for the context of the new line. We recommend to provide the context of the comment together with the suggested recommendation. The recommendation could be in the following form: "Previously, the reviewers identified a similar line <similar line from the training set> and provided the following comment <comment from the training set>."

6.8 Visualization of the results

In the visualization, the most important part is to show which lines of code the reviewer should focus on. This can be done in different ways — augmenting the code with comments, highlighting the code in Gerrit, highlighting the code directly in the IDE or creating a separate report.

When introducing this tool to the organization we recommend to check the way in which the team wants to set it up. In our case, we use the dashboard selection model for this purpose [**?**, 18]. The presentation of results is a kind of report, so augmenting the code in the Gerrit tool is the best visualization. However, this has the potential of cluttering the view, as this information will add more colors/annotations to the already cluttered user interface.

Therefore, a simple report with the lines recommended is often sufficient for the introductory stage. The code which is used for that is presented in Fig. 6.12.

```
def formatColor(row):
    color = 'background-color: {}'.format('lightgrey' if row.Review == 'No' else 'lightcoral')
    return (color, color, color)

styled = classifiedLines.style.apply(formatColor, axis=1)
styled.hide_index()

with open('./gerrit_review_comments_visualized_2.html', 'w') as f:
    f.write('<H1 style="text-align:center;">Results of identifying code fragments for manual review<H1/>')
    f.write('</BR>')
    f.write(styled.render())
```

Fig. 6.12: Code to visualize the results as an HTML report.

We also recommend to conduct an assessment of which kind of algorithms and types of machine learning should be adopted for the particular company [19].

6.9 Full example

In the remaining of this chapter, let us explore one example of classification and discuss the results. The code for our tool can be found at `https://github.com/miroslawstaron/auto_code_reviewer` on GitHub. In this example, we use the open source project Wireshark. The repository is available at `https://code.wireshark.org/review`.

The goal is to illustrate the output of each of the steps of the analysis and discuss them, not to obtain a perfect classification. We discuss the design decisions that can affect the performance of the classifiers as we proceed with the example.

The first step is the export. By executing the code exporting the data from the wireshark repository, we obtain the following output, as a `.csv` file. The head of the file is presented in Table 6.1.

The table contains lines of code that were commented and their corresponding comments. The first and the third comments are asking discussing the solution, whereas the second one is a confirmation that something has been done as a response to the comment.

Once we exported the raw data, we need to process it — use sentiment analysis on the comment messages and bag of words for source code lines. The application of the sentiment analysis for the comments results in the following sentiment results, presented in Table 6.2.

In order to validate the sentiment analysis, we manually labelled comments and compared that to the sentiment analysis. Figure 6.13 presents a confusion matrix and the accuracy scores for the validation. The high accuracy score (0.91) shows that even the simplistic, keyword-based sentiment analysis is rather good for our purposes.

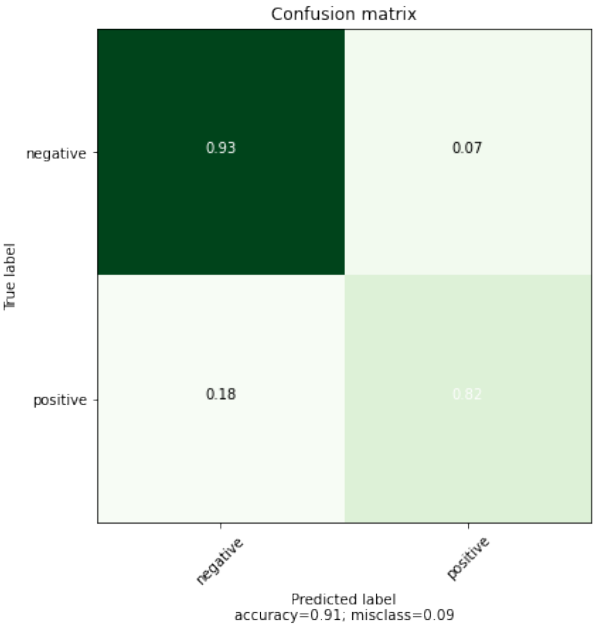


Fig. 6.13: Confusion matrix for the validation of sentiment analysis.

Table 6.1: Raw data exported from gerrit.

ch–id	rev–id	filename	line	startL	endL	LOC	message
1	r1	epan/dissectors packet-tls-utils.c	6061	0	0	tvb, offset, next_offset – offset, [truncated])	since you are not adding any extra information here, perhaps drop the [truncated] text here and rely on the label of the field in packet-tls-utils.h. And use proto_item_set_generated to get the [and] effect.
2	r2	epan/dissectors packet-tls-utils.c	6061	0	0	tvb, offset, next_offset – offset, "[truncated]")	Done
3	r3	epan/dissectors/ packet-afs.c	420	0	0	" char *version_type_"	const char * _ _ Can you also squash some trivial patches? Changing a typo in a comment can probably be done while you are modifying other code.

Table 6.2: Results of the sentiment analysis.

Message	Sentiment
since you are not adding any extra information here, perhaps drop the [truncated] text here and rely on the label of the field in packet-tls-utils.h. And use proto_item_set_generated to get the [and] effect.	0
Done	1
const char * _ _ Can you also squash some trivial patches? Changing a typo in a comment can probably be done while you are modifying other code.	1

Table 6.3: Bag of words features per line.

line	(tab)	(space)	!	“	#	$	%	...	a	able
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	0	0
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	0	0
char *version_type_”	1	1	0	0	0	0	0	...	0	0

Table 6.4: Raw data exported from gerrit.

line	(tab)	(space)	!	“	#	$	%	...	class_value
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	0
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	1
char *version_type_”	1	1	0	0	0	0	0	...	1

In parallel to the sentiment analysis, we also analyze the code and create the feature vector based on the bag of words analysis. The results are presented in Table 6.3. The table contains an excerpt of all the features identified, as the algorithm has identified 633 features.

Once we have both the sentiment for each comment and the feature vector for the line, we can merge them and create a matrix which we can use in the next step to train a machine learning model. The results are presented in Table 6.4.

Now that we have the lines labelled, we can use the AdaBoost algorithm to classify the lines. However, let us first create a diagram where we explore whether the classes are balanced, i.e. whether there is the same number of positive and negative instances. Figure 6.14 presents the results.

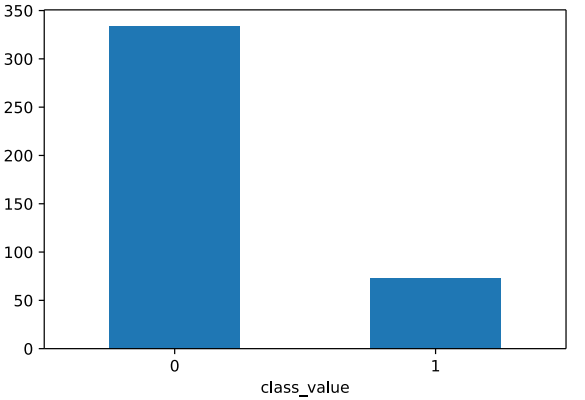


Fig. 6.14: Number of instances per class.

The diagram shows that the positive comments are fewer than the negative ones (which is quite common in code reviews). Therefore, we need to balance the classes, which is part of the toolchain described in this chapter.

Finally, we can train the machine learning classifier. The results of the trained classifier can be presented in different ways, here as a confusion matrix in Fig. 6.15.

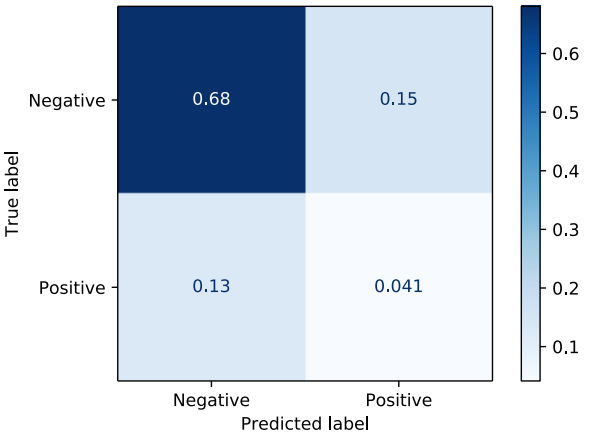


Fig. 6.15: Confusion matrix.

The results show that the classifier predicts most lines to be negative,

i.e. that they require manual reviews. This is caused by the example data set. We used a small data set for this example — ca. 400 lines — which is quite small for the classification. Since the code base of Wireshark significantly larger, over 2.8 million lines in different languages, using 400 lines to train the classifier is too little in practice, but ok for the illustration of the problem. To make it work in practice, the readers should modify the code provided in `https://github.com/miroslawstaron/auto_code_reviewer` to export all comments and then train the classifier. The results will be much better, i.e. more accurate.

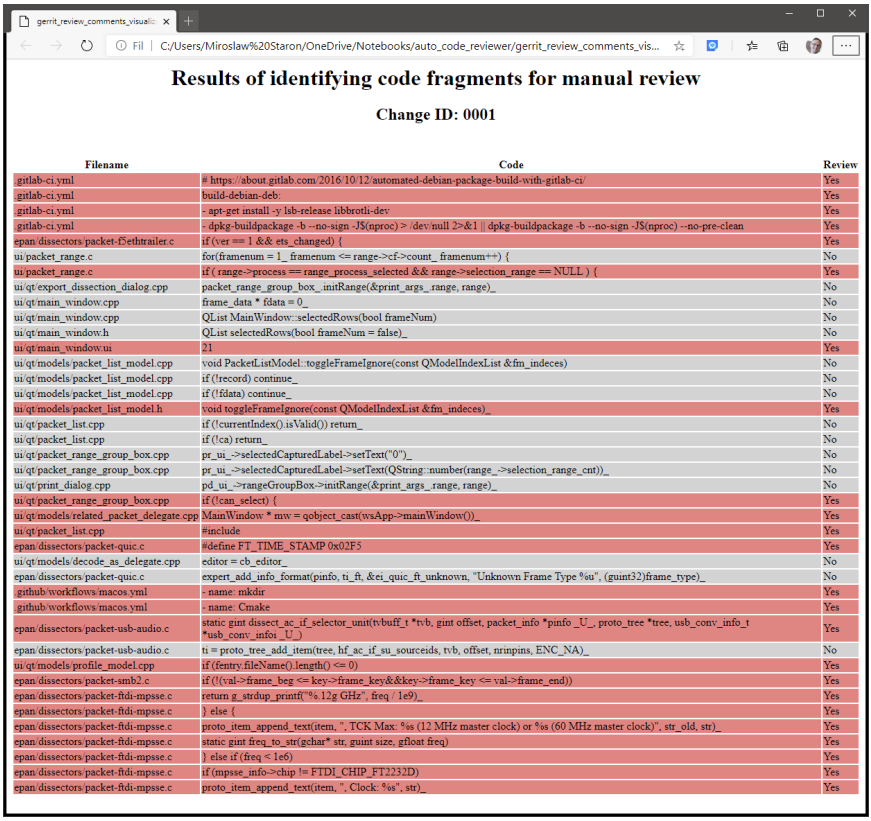
We can present the results as shown in Fig. 6.16. The figure presents an example of how this is visualized in an off-line manner, i.e. outside of the Gerrit review tool. Our future work is to create a plug-in to Gerrit to be able to highlight the lines for review in the Gerrit's user interface.

Presenting the results outside of the Gerrit's user interface is useful for the companies as this provides them with the possibility to review the entire code base, not only focus on a single patch. This helps the presented technology to get wider applicability than CI.

6.10 Using other techniques in the workflow

The workflow presented in this chapter, and depicted in Fig. 6.3, can be customized. We can exchange the source system, the feature extraction techniques and the classifiers. In our studies, we analyzed the following customizations:

(1) Exchanging the source tools with GitLab — we modified the export raw data script to use the GitLab API. The sentiment classification, bag-of-words and the classifier were not modified.
(2) Exchanging the feature extraction with own feature extraction techniques. We can exchange the bag-of-words with extractors that base on the keywords (e.g. `if`), tokens (e.g. `a0` instead of all variables that are named with small letters and numbers — `variable1` or `temp3`). We can even use more advanced techniques like word embeddings, but then we need to change the classifier to be a neural network.
(3) Exchanging the classifiers with neural networks. In the presented workflow we used the decision trees, but we can use more advanced algorithms. We can, for example, use convolutional neural networks. The advantage with deep learning techniques is that we can scale up the



Results of identifying code fragments for manual review

Change ID: 0001

Filename	Code	Review
gitlab-ci.yml	# https://about.gitlab.com/2016/10/12/automated-debian-package-build-with-gitlab-ci/	Yes
gitlab-ci.yml	build-debian-deb:	Yes
gitlab-ci.yml	- apt-get install -y lsb-release libbrotli-dev	Yes
gitlab-ci.yml	- dpkg-buildpackage -b --no-sign -J$(nproc) > /dev/null 2>&1 \|\| dpkg-buildpackage -b --no-sign -J$(nproc) --no-pre-clean	Yes
epan/dissectors/packet-f5ethtrailer.c	if (ver == 1 && ets_changed) {	Yes
ui/packet_range.c	for(framenum = 1_ framenum <= range->cf->count_ framenum++) {	No
ui/packet_range.c	if (range->process == range_process_selected && range->selection_range == NULL) {	Yes
ui/qt/export_dissection_dialog.cpp	packet_range_group_box_.initRange(&print_args_.range, range)_	No
ui/qt/main_window.cpp	frame_data * fdata = 0_	No
ui/qt/main_window.cpp	QList MainWindow::selectedRows(bool frameNum)	No
ui/qt/main_window.h	QList selectedRows(bool frameNum = false)_	No
ui/qt/main_window.ui	21	Yes
ui/qt/models/packet_list_model.cpp	void PacketListModel::toggleFrameIgnore(const QModelIndexList &fm_indeces)	No
ui/qt/models/packet_list_model.cpp	if (!record) continue_	No
ui/qt/models/packet_list_model.cpp	if (!fdata) continue_	No
ui/qt/models/packet_list_model.h	void toggleFrameIgnore(const QModelIndexList &fm_indeces)_	Yes
ui/qt/packet_list.cpp	if (!currentIndex().isValid()) return_	No
ui/qt/packet_list.cpp	if (!ca) return_	No
ui/qt/packet_range_group_box.cpp	pr_ui_->selectedCapturedLabel->setText("0")_	No
ui/qt/packet_range_group_box.cpp	pr_ui_->selectedCapturedLabel->setText(QString::number(range_->selection_range_cnt))_	No
ui/qt/print_dialog.cpp	pd_ui_->rangeGroupBox->initRange(&print_args_.range, range)_	No
ui/qt/packet_range_group_box.cpp	if (!can_select) {	Yes
ui/qt/models/related_packet_delegate.cpp	MainWindow * mw = qobject_cast(wsApp->mainWindow())_	Yes
ui/qt/packet_list.cpp	#include	Yes
epan/dissectors/packet-quic.c	#define FT_TIME_STAMP 0x02F5	Yes
ui/qt/models/decode_as_delegate.cpp	editor = cb_editor_	No
epan/dissectors/packet-quic.c	expert_add_info_format(pinfo, ti_ft, &ei_quic_ft_unknown, "Unknown Frame Type %u", (guint32)frame_type)_	No
.github/workflows/macos.yml	- name: mkdir	Yes
.github/workflows/macos.yml	- name: Cmake	Yes
epan/dissectors/packet-usb-audio.c	static gint dissect_ac_if_selector_unit(tvbuff_t *tvb, gint offset, packet_info *pinfo _U_, proto_tree *tree, usb_conv_info_t *usb_conv_info _U_)	Yes
epan/dissectors/packet-usb-audio.c	ti = proto_tree_add_item(tree, hf_ac_if_su_sourceids, tvb, offset, nrinpins, ENC_NA)_	No
ui/qt/models/profile_model.cpp	if (fentry.fileName().length() <= 0)	Yes
epan/dissectors/packet-smb2.c	if (!(val->frame_beg <= key->frame_key&&key->frame_key <= val->frame_end))	Yes
epan/dissectors/packet-ftdi-mpsse.c	return g_strdup_printf("%.12g GHz", freq / 1e9)_	Yes
epan/dissectors/packet-ftdi-mpsse.c	} else {	Yes
epan/dissectors/packet-ftdi-mpsse.c	proto_item_append_text(item, ", TCK Max: %s (12 MHz master clock) or %s (60 MHz master clock)", str_old, str)_	Yes
epan/dissectors/packet-ftdi-mpsse.c	static gint freq_to_str(gchar* str, guint size, gfloat freq)	Yes
epan/dissectors/packet-ftdi-mpsse.c	} else if (freq < 1e6)	Yes
epan/dissectors/packet-ftdi-mpsse.c	if (mpsse_info->chip != FTDI_CHIP_FT2232D)	Yes
epan/dissectors/packet-ftdi-mpsse.c	proto_item_append_text(item, ", Clock: %s", str)_	Yes

Fig. 6.16: Number of instances per class.

feature vectors to very large, we can also process much larger data sets (we experimented with products over 10 million LOC).

(4) Exchanging the sentiment analysis method. We used a simplistic sentiment analysis method, which is 91% accurate. However, it has its limitations and cannot capture sophisticated comments. To improve that, we can use natural networks for sentiment analysis and capture the details of the comments.

We recommend the companies and researchers to experiment with own techniques to get the most out of the presented work. The code base is provided in GitHub as open source under the GPL v.3 license.

6.11 Related work

The ideas of mining software data from existing repositories was populated by the seminal work of Zimmermann *et al.* [20]. The authors demonstrated how using past software design data, mined from software repositories, could help to improve the quality of software.

Liang *et al.* [21] expanded the work of Zimmermann et al. and analyzed patterns of involvement of designers in code reviews. The work opened up important alleys, but was not done in the context of continuous integration, which is the focus of our work. Bernhard *et al.* [22, 23] indicated that the Agile software development introduces changes in the way that software reviews work (compared to the non-Agile and non-CI ways of working).

Chatley and Jones [24] developed a tool for generating review comments, which is similar to what we wanted to achieve. However, the major difference is that we provide the designers with the freedom to interpret the code fragments, not generating the reviews. The important difference is that we provide the possibility for the organizations to learn, i.e. taking into consideration the human factors [25].

These human aspects are important when considering code reviews in larger industrial contexts [26]. There, learning and understanding code is important, as well as the need to discussions about the design solutions in the code. In such contexts even the softer aspects are important, e.g. code ownership [27].

Our work on data management is follows similar principles as the work of Menzies *et al.* [28]. Menzies *et al.* provided a comprehensive work on the use of big data in software engineering. We complement their work by contributing with a modern method for extracting code reviews from automated tools like Gerrit.

Another important position, which our research complement, is the work by Bird *et al.* [29], which focuses on mining software repositories. We can consider Gerrit to be one type of software repository and our methods help to mine the data, analyze it automatically, and use the new knowledge to help software engineers in their work.

In the area of code review, Pascarella *et al.* [30] studied what kind of needs the designers have in modern code reviews. The results show that the designers need to understand the usage of methods and design guidelines. Our work contributes to the automation of the review process and therefore complements this work, but opening up for automated matching between code fragments and comments.

Ebert *et al.* [31] studied the code reviews in more depth and observed a similar trend as we — that the comments often can be misinterpreted. The work of Ebert can be used to reduce the ambiguity in the sentiment analysis and therefore lead to more accurate results of the code classification, thus complementing our work.

Finally, a recent systematic mapping from Badampudi *et al.* [32] indicate that the current state-of-the-art in this area is focused on the tool support for the review process, but not in the automation of the code review, where our work fits.

6.12 Conclusions

Machine learning and artificial intelligence has risen in popularity for the past few years. These methods, which allow for automated handling of large quantities of data, seem to be a perfect match for the modern challenges of software engineering. In this chapter, we addressed one of the challenges in modern software engineering industry — achieving high quality software in the high velocity continuous integration pipelines.

By using machine learning, we designed a system that can learn from previous code reviews by abstracting both the reviewed code and the comments. The system illustrates how convenient machine learning is to solve software engineering problems. It also illustrates how to work with the data management workflows in this context.

Out further work is based on the defining a specific language model for comments in software engineering, which can provide the ability to automatically analyze the reviews with more accuracy.

References

[1] P. M. Duvall, S. Matyas and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education (2007).

[2] M. Meyer, Continuous integration and its tools, *IEEE software* **31**, 3, pp. 14–16 (2014).

[3] K. Al-Sabbagh, M. Staron, R. Hebig and W. Meding, Predicting test case verdicts using textual analysis of commited code churns, (2019).

[4] A. Debbiche, M. Dienér and R. B. Svensson, Challenges when adopting continuous integration: A case study, in *International Conference on Product-Focused Software Process Improvement*. Springer, pp. 17–32 (2014).

[5] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon and C. Jaspan, Lessons from building static analysis tools at google, (2018).

[6] F. Zampetti, G. Bavota, G. Canfora and M. Di Penta, A study on the interplay between pull request review and continuous integration builds, in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 38–48 (2019).

[7] ZDNet, Linus Torvalds: I'm not a programmer anymore, *ZDNet magazine*, (2019).

[8] M. Ochodek, R. Hebig, W. Meding, G. Frost and M. Staron, Recognizing lines of code violating company-specific coding guidelines using machine learning, *Empirical Software Engineering* **25**, 1, pp. 220–265 (2020).

[9] L. Milanesio, *Learning Gerrit Code Review*. Packt Publishing Ltd. (2013).

[10] M. Staron, *Action Research in Software Engineering: Theory and Applications*. Springer Nature (2019).

[11] I. Fronza, A. Hellas, P. Ihantola and T. Mikkonen, Code reviews, software inspections, and code walkthroughs: Systematic mapping study of research topics, in *International Conference on Software Quality*. Springer, pp. 121–133 (2020).

[12] M. Staron, W. Meding, O. Söder and M. Bäck, Measurement and impact factors of speed of reviews and integration in continuous software engineering, *Foundations of Computing and Decision Sciences* **43**, 4, pp. 281–303 (2018).

[13] H. H. Olsson, H. Alahyari and J. Bosch, Climbing the "stairway to heaven" — a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software, in *2012 38th euromicro conference on software engineering and advanced applications*. IEEE, pp. 392–399 (2012).

[14] M. Ochodek, M. Staron, D. Bargowski, W. Meding and R. Hebig, Using machine learning to design a flexible loc counter, in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, pp. 14–20 (2017).

[15] M. Taboada, J. Brooke, M. Tofiloski, K. Voll and M. Stede, Lexicon-based methods for sentiment analysis, *Computational linguistics* **37**, 2, pp. 267–307 (2011).

[16] P. H. Swain and H. Hauska, The decision tree classifier: Design and potential, *IEEE Transactions on Geoscience Electronics* **15**, 3, pp. 142–147 (1977).

[17] C. Ying, M. Qi-Guang, L. Jia-Chen and G. Lin, Advance and prospects of adaboost algorithm, *Acta Automatica Sinica* **39**, 6, pp. 745–758 (2013).

[18] M. Staron, K. Niesel and W. Meding, Selecting the right visualization of indicators and measures–dashboard selection model, in *Software Measurement*. Springer, pp. 130–143 (2015).

[19] R. Rana, M. Staron, J. Hansson, M. Nilsson and W. Meding, A framework for adoption of machine learning in industry for software defect prediction, in *2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA)*. IEEE, pp. 383–392 (2014).

[20] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl, Mining version histories to guide software changes, *IEEE Transactions on Software Engineering* **31**, 6, pp. 429–445 (2005).

[21] J. Liang and O. Mizuno, Analyzing involvements of reviewers through mining a code review repository, in *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*. IEEE, pp. 126–132 (2011).

[22] M. Bernhart, A. Mauczka and T. Grechenig, Adopting code reviews for agile software development, in *2010 Agile Conference*. IEEE, pp. 44–47 (2010).

[23] M. Bernhart and T. Grechenig, On the understanding of programs with continuous code reviews, in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, pp. 192–198 (2013).

[24] R. Chatley and L. Jones, Diggit: Automated code review via software repository mining, in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 567–571 (2018).

[25] S. Ruangwan, P. Thongtanunam, A. Ihara and K. Matsumoto, The impact of human factors on the participation decision of reviewers in modern code review, *Empirical Software Engineering* **24**, 2, pp. 973–1016 (2019).

[26] Y. Tao, Y. Dang, T. Xie, D. Zhang and S. Kim, How do software engineers understand code changes? an exploratory study in industry, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11 (2012).

[27] L. P. Hattori, M. Lanza and R. Robbes, Refining code ownership with synchronous changes, *Empirical Software Engineering* **17**, 4–5, pp. 467–499 (2012).

[28] T. Menzies, L. Williams and T. Zimmermann, *Perspectives on data science for software engineering*. Morgan Kaufmann (2016).

[29] C. Bird, T. Menzies and T. Zimmermann, *The art and science of analyzing software data*. Elsevier (2015).

[30] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink and A. Bacchelli, Information needs in contemporary code review, *Proc. ACM Hum.-Comput. Interact.* **2**, CSCW (2018), doi:10.1145/3274404, `https://doi.org/10.1145/3274404`.

[31] F. Ebert, F. Castor, N. Novielli and A. Serebrenik, Confusion in code reviews: Reasons, impacts, and coping strategies, in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 49–60 (2019).

[32] D. Badampudi, R. Britto and M. Unterkalmsteiner, Modern code reviews-preliminary results of a systematic mapping study, in *Proceedings of the Evaluation and Assessment on Software Engineering*, pp. 340–345 (2019).

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0007

Chapter 7

Software Fusion: Deep Design Learning with Deterministic Laplacian Verification

Iaakov Exman

Department of Software Engineering
The Jerusalem College of Engineering – JCE – Azrieli,
Jerusalem, Israel
iaakov@jce.ac.il

7.1 Introduction

Automatic software generation is potentially a powerful tool to assure software correctness. It could be of utmost importance for critical systems, to avoid accidents with dire material consequences and human life losses. As a recent example, airplanes of the Boeing 737 Max 8 type were grounded after two accidents with hundreds of fatalities. The accident causes in the embedded avionics software, were not amenable to correction within a short time.

The large investments in driverless autonomous vehicles, by cars and trucks industries, are another example of the growing importance of software correctness in everyday life. They are being justified by the higher reliability of software driven steering relative to human drivers. This has not been yet achieved, but potentially it could be true.

Software correctness is also important for non-safety-critical systems. Very large software systems used on a routine basis, such as transaction payments and travel reservation systems, among others, must be bug-free, for evident reasons.

7.1.1 *Software Design Generation instead of Lower-Abstraction Assets*

The overwhelming majority of previous research efforts to generate software by Deep Learning neural networks have been directed towards lower-level abstraction software implementation assets, such as runnable code and comments. These efforts focused on lower-level abstraction software assets because the needed neural networks architectures — mainly sequential RNN (Recurrent Neural Networks) variants — were available and easily applied. Programming language statements are indeed sequential. But the obtained results were fragile. They are not easily amenable to robust correctness, due to the probabilistic nature of the neural nets, followed by traditional trial-and-error human debugging.

This chapter describes *Software Fusion*, a novel and original approach, aiming at the *higher-abstraction Software Design*. It is based upon more flexible GCN (Graph Convolutional Networks), whose graph structures can represent any software system, at the design level. For instance, UML class diagrams can be viewed as graphs, with nodes standing for classes and edges standing for any relationships such as composition or inheritance, in object-oriented parlance.

Rigorous verification of graphs obtained by Deep Learning is essential to attain software correctness. Higher-level abstraction is robust due to *deterministic* correctness verification. It uses spectral methods upon Laplacian matrices — an algebraic software design representation within Linear Software Models [11] — fitting the respective graphs.

Software Fusion is Software system design done by fusion of two complementary mathematical tools, as shown in Fig. 7.1:

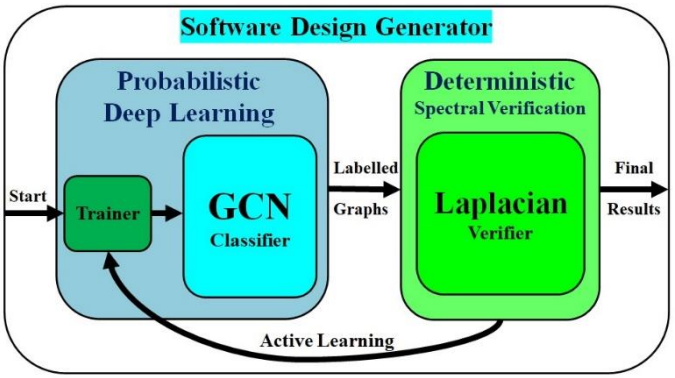


Fig. 7.1. Software Fusion — Software Design Schematic Architecture — It consists of the interaction of two complementary mathematical tools. In the picture left-hand-side (blue background) one sees the 1st Tool — Probabilistic Deep Learning, including the GCN classifier and its preceding trainer. In the picture right-hand-side (green background) one sees the 2nd Tool — Deterministic Laplacian matrix Spectral verification. Final Results contain Graphs verified to be correctly labelled. Graphs incorrectly labelled by the GCN classifier, can be corrected, and reused to improve training, by "Active Learning".

- 1st Tool — ***Probabilistic Deep Learning generation of design graphs*** by GCN — it starts with training of the GCN with a suitable number of a priori labelled graphs. Then the GCN classifies test cases, outputting their labelled graphs.
- 2nd Tool — ***Deterministic Laplacian matrix Spectral Verification*** of the output of the 1st Tool. If the latter labelled graphs are correctly verified these are included in the Final Results of software design generation. If not, mistaken labels can be corrected and reused to improve the GCN training, in an "active learning" [35] fashion.

7.1.2 *Experimental Results Obtained*

Modularity is a basic concept of well-designed software systems. A whole software system is hierarchically built from modular — i.e. well-defined and separable — sub-systems, which in turn are built from modular sub-sub-systems, and so on, down to architectural units assumed indivisible by the software engineer, typically classes in object-oriented design.

The purpose of Graph Convolutional Networks experiments is:

- to test that the labelling behavior of the GCN classifier had the *expected functional dependence* on the input parameters of the experiments. For instance, classifier labelling accuracy improves with increasing sizes of the modules. Sensible functional dependence is essential for successful deterministic verification and automated correction of software design.

Results of the experiments with the GCN classifier, demonstrate: (a) the classifier dependence on the input parameters fits to the expected functionality; (b) the classifier distinguishes well-designed modular software systems from software design generated systems of lesser quality, e.g. having undesirable "*outlier*" units coupling potential modules.

The Laplacian Spectral Verifier experiments purpose was to show, beyond its theoretical correctness [7], [17], its applicability in practice to software design. Experiments submitted the Laplacian Verifier to a wide variety of sub-systems, with and without outliers, to test its reliability.

7.1.3 *Chapter Organization*

This chapter argues for higher-abstraction to generate software design by Deep Learning, aiming at software correctness. It is organized as follows:

- **Section 7.2** points out to Deep Learning fragility of sequential lower-level software assets, as program code, comments, and documentation. Typically, these use RNN (Recurrent Neural Networks) [40] variants.
- **Section 7.3** explains the source of design robustness, due to Linear Software Models [14], detailing the steps from an UML class diagram to the deterministic spectral Laplacian verification of software design.
- **Section 7.4** introduces Graph Convolutional Networks (GCN) and specializations applied to higher-abstraction levels of software design. These are suitable for bipartite graphs derived from class diagrams.
- **Section 7.5** illustrates software system modular design using GCN, by means of Case Study experiments and their results.
- **Section 7.6** mentions Related Work relevant to this chapter topics.
- **Section 7.7** ends the chapter with a Discussion of foundational issues.

7.2 Fragility of Lower-Abstraction Software Deep Learning

Section 7.2 points out the fragility of the Deep Learning approach toward *lower-abstraction software assets*. We explain why this is not a suitable level to automatically generate *correct* software assets.

7.2.1 *Lower Abstraction Levels of Software Generation*

We concisely survey lower-abstraction Deep Learning generation of software assets. These assets have a sequential textual nature. The survey is in increasing software semantic content, bottom-up in Fig. 7.2.

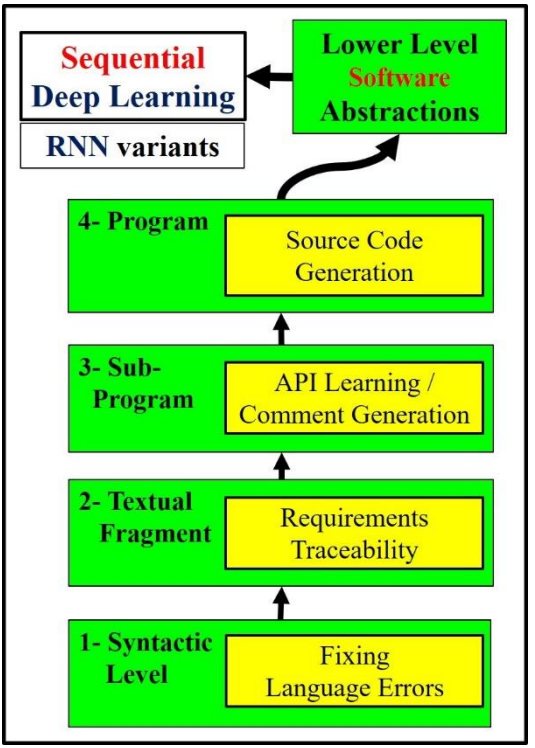


Fig. 7.2. Lower-Level Software Abstractions Hierarchy — Deep Learning for lower-level abstractions is done sequentially by RNN variants. These abstractions (on green background) have increasing semantic content in the figure bottom-up direction. Typical application examples are shown beside each abstraction-level (on yellow background).

Sequential textual applications are based upon RNN (Recurrent Neural Networks) [40] models and their most common variants — e.g. LSTM (Long Short-Term Memory) [26], and GRU (Gated Recurrent Unit) [5].

The software semantic content levels are the following:

(1) ***Syntactic Level*** — *Fixing Language Errors* is a task at the bottom abstraction level in the Fig. 7.2 hierarchy. It corrects common programming errors, such as missing scope delimiters, e.g. a closing brace, only depending on the programming language syntax. Fixing these errors does ***not*** require ***semantic knowledge*** of the ***task***.

Deepfix by Gupta *et al.* [23] is an example for this task. Its features are:

- It aims at *automatically* fixing programming errors, which is a laudable goal, aligned with our automation intention;
- It treats a program as a *sequence of tokens*;
- Its architecture is a multi-layered *sequence-to-sequence* neural network with attention, i.e. an encoder RNN processes erroneous inputs and a decoder RNN with attention generates outputs with error corrections;
- In order to increase efficiency, it uses the peculiar solution of *encoding program line-numbers* in the program representation.

(2) ***Textual Fragment level*** — the next software assets generation task, going upward in the hierarchy of Fig. 7.2, involves textual fragments. These fragments do not refer to a software architectural unit. It essentially is an NLP (Natural Language Processing) task.

Requirements Traceability is a typical task of this kind. Its challenge is to recognize links between pairs of items belonging to different software assets, e.g. requirements, faults, design, source code and test cases. It requires non-trivial domain concepts knowledge in the relevant items, i.e. more semantic content than the strictly Syntactic Level tasks. But it still *does not demand semantics knowledge of architectural units*.

An example is software traceability by Jin Guo *et al.* [22], featuring:

- Its network architecture is *multi-layered RNN variants* with *Word Embedding*. Their best Traceability model is the BI-GRU (Bidirectional Gated Recurrent Unit) variant;

- Words in the source and target artifacts — the item pairs to be linked — are sent to the RNN layers ***sequentially***, and semantic vectors are generated and compared;
- A usual trait of software projects is the very small number of actual trace links compared to the number of the potential item pairs;
- *Automated traceability solutions* are desirable to cope with a time consuming and error-prone task; but data preparation uses too large corpora of domain documents.

(3) ***Sub-Program Level*** — is a software assets generation task of sub-programs, e.g. Java methods. It involves analysis of *code snippets*, which are *not yet a whole integrated software* system. Each code snippet certainly has more software semantic integrity than tasks involving syntactic errors or tracing pairs of isolated items.

An example of this task is *API Extraction*: a programmer wishes to use an API function, but either does not know which function is relevant to his problem, or knows which API function, but does not know how to use it.

A sample API Extraction application is DeepAPI by Xiaodong Gu *et al.* [21]. DeepAPI gets from the inquiring programmer a natural language query and replies with one or more API usage sequences, each one supplemented with a short annotation. Its characteristics are:

- it extracts <API usage sequence | natural language annotation> pairs from a code corpus (of GitHub Java projects); the usage sequence is derived from ASTs (Abstract Syntax Trees) and a Javadoc annotation;
- The <*API sequence | annotation*> pairs are used to train a Deep Learning model, viz. a GRU variant of an RNN Encoder-Decoder;
- Instead of a bag-of-words DeepAPI learns word sequences, mapping them to semantic vectors. Similar words obtain similar vectors;
- Finally, the RNN Encoder-Decoder is ready to translate API-related programmer queries into suggested and ranked API sequences.

Another sub-program level task example is *Comment Generation*. It is similar to API extraction, but in the opposite direction. The input to API extraction is a natural language query and it outputs an API sub-program.

On the other hand, the input to Comment Generation is a sub-program, e.g. a function, and its output is a short natural language comment.

A recent sample of Comment Generation by Deep Learning is the DeepCom system described in the work of Xing Hu *et al.* [27].

(4) ***Program Generation*** is the top level in the Fig. 7.2 hierarchy. This more ambitious task is to generate program source code from certain kinds of input. It is supposed to *automatically generate a complete program* — perhaps not very long, but still a complete program — not just code snippets. (See also DeepSoft, by Dam *et al.*, [6].)

A program generation example is in a paper by Mou *et al.* [38]. Its input is a brief comment such as "find maximal number" (ended by the special symbol <eos> "end-of-sequence"). Its features are:

- Code is generated in a *symbol-by-symbol fashion*; the symbol granularity may be a word, sub-word, or a single character (as in [38]);
- RNN does *sequence to sequence generation*; a softmax layer at each time step, predicts symbol occurrence probabilities; the highest probability symbol is fed to the network input at the next time step;
- A network is *trained by* e.g. *hundreds of different source code samples* for each program task; all samples must be correctly compiled and run.

7.2.2 *Fault Lines of Lower Software Abstractions*

Here are the main issues with Sequential Lower-abstraction software assets Deep Learning, as illustrated by the examples in Sec. 7.2.1.

(A) **Brittleness of lower-abstraction Deep Learning** — To what extent can it generate Bug-Free software assets?

For instance, the results of Deepfix are far from satisfactory in terms of correctness. According to the paper by Gupta *et al.* [23] it completely fixed only 27% of the erroneous programs and partially fixed only 19% of the programs of a "representative" data set.

A related correctness obstacle, is illustrated by the Traceability task in the work by Jin Guo *et al.* [22], viz. data preparation uses large corpora of

domain documents; the scenario for training the tracing network, uses an initial set of *manually constructed trace links* for specific projects.

In preliminary results of the Program Generation by Mou *et al.* [38], the *generated program is "almost" correct*. A generated code example in their paper [38] has about 15 lines of code and 4 one-symbol errors:

(i) Mistaken identifiers — an "***x***" should have been an "***n***";
(ii) Misspelled identifiers — "***max***" should have been "***max2***";
(iii) Wrong symbol — "**==**" should have been "<";
(iv) Wrong return type — it should have been "***void***";

An experienced programmer could easily debug four minor errors. As the paper states: code is usable with little post-editing. But the approach is unrealistic for real size programs, e.g. tens of thousands of lines of code.

These errors are semantically uninteresting. One can correct them, but one does not learn anything much valuable from these errors: they are too devoid of content, to enable some positive uses of these mistakes.

(B) **Conceptual-Disconnect** — Can lower-abstraction software assets Deep Learning overcome the algebra and semantics Disconnect?

"Impossibility to completely decipher how the semantic vector is extracted" — this perception (last paragraph, 1st column, page 8) in Guo's Traceability paper [22], illustrates a severe algebraic/semantic disconnect.

Mou *et al.* [38], generate programs in a *symbol-by-symbol fashion*, with *single character symbols*. The latter do not convey semantics, being an impediment to deeper understanding of the input/output relationship.

Summarizing this section, sequential lower-abstraction Deep Learning does not attain the purpose of software assets generation *automation*. The fragility of lower-abstraction software generation has two causes:

- *low-level bugs* arduously corrected by human trial-and-error; low-level bugs are inherent to the probabilistic nature of Deep Learning of low-abstraction models, difficult to solve in a fundamental way.
- the *conceptual disconnect* between algebra and semantics occurs because semantics is not naturally embedded in the inner computation steps of the Deep Learning algebra. (Cf. discussion in Sec. 7.7.1.)

7.3 Robustness of Software Fusion Design

This section shows how higher-abstraction solves lower-abstraction Deep Learning problems:

- ***deterministic design verification*** by a Laplacian spectral approach pinpoints mistaken design graphs assuring software design correctness;
- ***conceptual integrity*** is guaranteed by semantics embedded ab initio into the algebra and preserved throughout the algebraic manipulation of Linear Software Models.

The source of *Software Fusion* robustness is the deterministic Laplacian verifier (see Fig. 7.1 in Sec. 7.1.1 of this chapter).

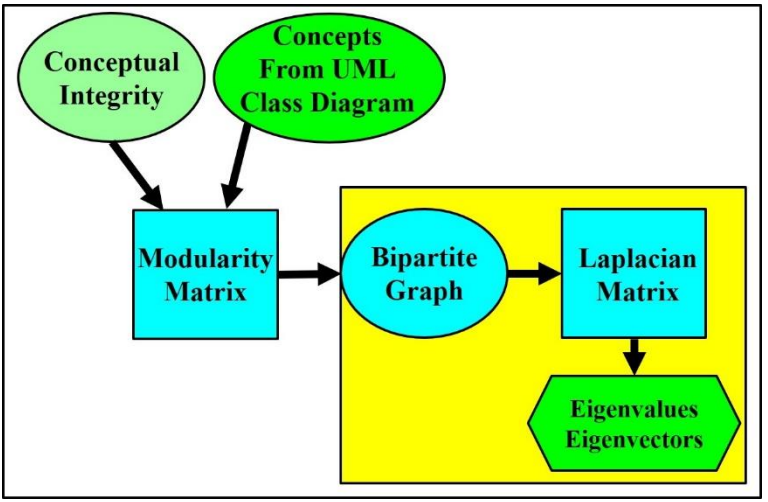


Fig. 7.3. Linear Software Models: entities and transitions — In the middle of this figure one sees three equivalent software system design representations: Modularity Matrix, Bipartite Graph and Laplacian Matrix (on blue background). The Modularity Matrix form is due to the Conceptual Integrity constraints (Sec. 7.3.1). Modularity Matrix concepts are obtained e.g. from a UML Class Diagram (Sec. 7.3.2). A bipartite graph is easily extracted from the Modularity Matrix (Sec. 7.3.3). The Laplacian Matrix is defined in terms of the bipartite graph (Sec. 7.3.4). The deterministic Laplacian Matrix verifier (on yellow background) relies upon the Laplacian eigenvalues and eigenvectors.

The sub-sections of Sec. 7.3 carefully explain how:

- *software system design is represented* by matrices and a bipartite graph;

- *representations are obtained*, e.g. from a UML class diagram;
- *semantics is embedded* in the linear algebra and preserved throughout algebraic manipulation of software design;
- *deterministic spectral verification* is based upon eigenvalues and eigenvectors of the Laplacian Matrix.

The transitions between the relevant entities — the matrices and the bipartite graph of Linear Software Models [14] — are collected in Fig. 7.3.

7.3.1 *Conceptual Integrity for Software Design*

Semantics is embedded in the Linear Software Models algebraic representation entities of software systems' design, through the constraints imposed by Conceptual Integrity. This is an idea first formulated by Frederick P. Brooks Jr. expressed as follows: "***Conceptual Integrity*** is the most important consideration in system design". This idea appeared in Brooks' well-known book "The Mythical Man-Month — Essays in Software Engineering", first published in 1974 [1]. It was based upon his extensive development experience of the first large software system of its time, the OS 360 Operating System of IBM.

To apply Conceptual Integrity in practice to software design, Brooks in his second book "The Design of Design" [2] proposed principles that could serve as design criteria. Two of Brooks' principles — here formulated in terms of functions instead of concepts [9] — are as follows:

- ***Propriety*** — a product should have only the *functions essential* to its purpose and no more;
- ***Orthogonality*** — individual *functions should be independent* of one another.

We represent software system design by Linear Software Models [10] based upon linear algebra. For instance, a Modularity Matrix [11] whose columns stand for *Structors*, generalizing object-oriented classes, and whose rows stand for *Functionals*, generalizing class methods. A matrix element is 1-valued if its *Structor* provides the respective *Functional*, otherwise it is zero-valued. *Provides* means that the class contains the definition/declaration of a method.

The purpose of software design is to optimize its Modularity. To this end we translated Brooks' principles to linear algebra terms [11]:

- ***Propriety*** — *Structor* column vectors should be mutually linearly independent and *Functional* row vectors should also be mutually linearly independent. Thus, *Propriety* minimizes classes and functions to just the ***essential ones***. Under these conditions, by pure linear algebra arguments, the Modularity Matrix is square. This is a theorem [11], not a trivial one, and it takes some time to understand its meaning.
- ***Orthogonality*** — this is a stronger demand than linear independence; if a Structors' set and their respective Functionals are disjoint to other Structors' sets and their Functionals, a Modularity Matrix can be put in block-diagonal form. The diagonal blocks are recognized as the software system modules: Structors and Functionals belonging to a given module are ***orthogonal*** to those vectors of all other modules.

Structors and Functionals are labelled by concepts. Concepts are carried throughout the algebraic manipulations, preserving the ***software system connect*** between algebra and semantics, whatever is the algebraic representation in use. (See the discussion in Sec. 7.7.1.)

7.3.2 *Modularity Matrix from UML Class Diagram*

Concepts of a software system design — labelling Structors and Functionals of the Modularity Matrix – are chosen according to the *human* software engineer *understanding* of the software system requirements.

Concepts can be obtained from many sources: (a) an UML Class Diagram; (b) the source code implemented in a programming language; (c) the program executable code, using the Modulaser [13] software tool.

The idea is illustrated starting from an UML class diagram of the Command design pattern (see the GoF Design Patterns' book [18]) as a running example. Figure 7.4 shows a class diagram of this design pattern. Its Modularity Matrix [6] is seen in Fig. 7.5. The numbers of Structor columns $\boldsymbol{S_i}$ and of Functional rows $\boldsymbol{F_i}$ are equal. It is a standard Modularity Matrix: it is square, has strictly block-diagonal modules, and no outliers outside modules. Inheritance in the class diagram is translated to a row with two or more 1-valued matrix elements in the Modularity Matrix.

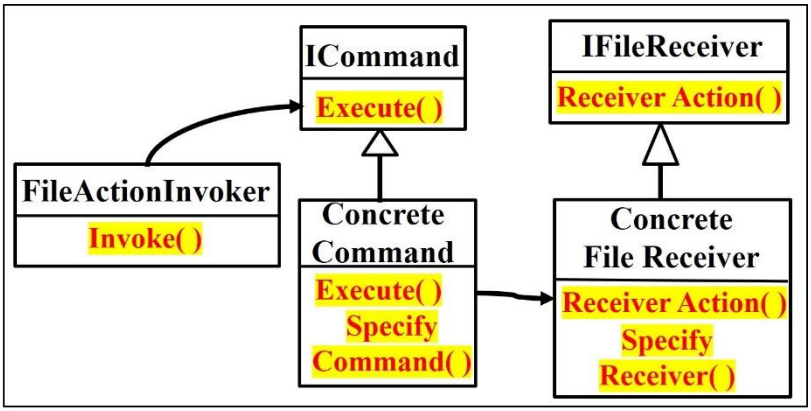


Fig. 7.4. Command Design Pattern UML Class Diagram — It has 2 interfaces ICommand and IFileReceiver, and 3 classes FileActionInvoker, ConcreteCommand and ConcreteFileReceiver (black names). Interfaces or classes have methods (red names on yellow). ConcreteCommand has methods SpecifyCommand( ) and Execute( ), inherited from the ICommand interface. The ConcreteCommand class is aware, as marked by a black arrow, of the ConcreteFileReceiver methods and can call the ReceiverAction( ) method.

Structors →		ICommand	Concrete Command	File Action Invoker	IFile Receiver	Concr.File Receiver
Functionals ↓		S1	S2	S3	S4	S5
Execute	F1	1	1			
Specify Command	F2	0	1		Zero-valued elements	
Invoke	F3			1		
Receiver Action	F4	Zero-valued elements			1	1
Specify Receiver	F5				0	1

Fig. 7.5. Command Design Pattern Modularity Matrix — has five Structors (S1, …, S5), five Functionals (F1, …, F5), and three (blue) block-diagonal modules: two 2*2 modules, and a middle 1*1 module. Zero-valued elements outside modules are omitted. Structors and Functionals have double-identifiers: semantic concepts and short labels S_i, F_i.

The modules shown in Fig. 7.5 use in advance results to be calculated a posteriori from the Laplacian Matrix, as explained in Sec. 7.3.4.

An important question is scalability with respect to the software system size. These algebraic representations are indeed scalable. This is achieved by the modular hierarchy assumption stated in the first paragraph of Sec. 7.1.3: a whole software system is hierarchically built from modular, i.e. separable sub-systems, which in turn are built from modular sub-sub-systems, and so on. Instead of very large matrices per system, one uses a hierarchy of smaller matrix sets, each set corresponding to a given level of the hierarchy. A module in a matrix in a given level, can be collapsed into a single matrix element in the immediate upper level matrix.

7.3.3 *Bipartite Graph from the Modularity Matrix*

The *Bipartite Graph* in Fig. 7.6 fits the Modularity Matrix in Fig. 7.5. It is obtained as follows: Structors are drawn as an upper set of vertices; Functionals are drawn as a lower set of vertices; each 1-valued matrix element in the Modularity Matrix generates one edge in the bipartite graph.

The graph in Fig. 7.6 is bipartite by definition [47]: it has two vertex sets, the upper Structors and the lower Functionals set; edges only link vertices in the opposite set, not in the same set.

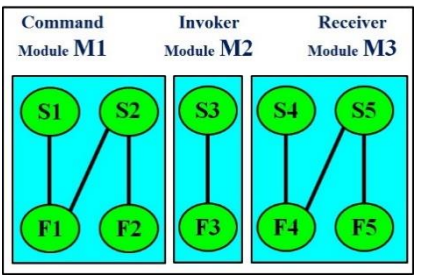


Fig. 7.6. Command Design Pattern Bipartite Graph fitting the Modularity Matrix in Fig. 7.5 — It has the same Structor and Functional vertices as the Matrix. Within Software Fusion, the modules are obtained a posteriori from the Laplacian Matrix.

Bipartite graphs, the transition between the Modularity Matrix and its corresponding Laplacian Matrix, are a cornerstone of our Deep Learning by *Graph Neural Networks*. The Bipartite Graph contains all the

information in the Modularity Matrix. Therefore, it is a good starting point and also a reasonable outcome for the GCN.

7.3.4 *Deterministic Spectral Verification of Software Design Graph*

Whether the software design bipartite graph is obtained by an experienced software engineer or by the intended automatic Deep Learning, the bipartite graph modularization can be rigorously verified with the deterministic spectral approach in four steps:

(a) ***Generate the Laplacian matrix*** from the Bipartite Graph;
(b) ***Calculate eigenvalues/eigenvectors*** of the Laplacian matrix;
(c) ***Obtain software system modules*** — eigenvectors with zero-valued eigenvalues obtain modules; modularization may take a few iterations as detailed below;
(d) ***Check for outlier Laplacian matrix elements*** — outliers highlight design problems to be solved; absence of outliers means successful software design.

For any graph with a total number N of vertices v_i where $i = 1, \ldots, N$, the definition of its Laplacian matrix L is given by the following equation:

$$L = D - A \tag{7.1}$$

where D is a diagonal Degree matrix and A is the Adjacency matrix. The Degree matrix contains in its matrix elements the degree of each of the vertices in the graph. The Adjacency matrix contains in its non-zero matrix elements, a pointer to neighbors of each vertex.

Before presenting the spectral verification algorithm, two notions are needed: *Fiedler Vector* and *Module Sparsity*.

- *The Fiedler vector* — this eigenvector named after the mathematician Fiedler [17], [7], fits the lowest positive eigenvalue of the Laplacian. The Fiedler vector enables splitting a module judged to be too large. Splitting into two smaller modules uses the non-zero vector elements: one module by the positive eigenvector elements and another module by the negative eigenvector elements.
- *Module Sparsity* — Software modules have denser internal connections than inter-module connections. This is the mathematical expression of

module Cohesion. When an outlier couples between two "actual" modules, they are seen by the Laplacian eigenvectors as a single large module containing the two "actual" modules and the outlier: an outlier causes a bigger module, recognizable by its Sparsity. A too sparse module must be split by the Fiedler vector, and the outlier refactored.

The Laplacian fitting the bipartite graph of the Command pattern in Fig. 7.6 is seen in Fig. 7.7.

	F1	F2	F3	F4	F5	S1	S2	S3	S4	S5
F1	2					-1	-1			
F2		1				0	-1			
F3			1					-1		
F4				2					-1	-1
F5					1				0	-1
S1	-1	0				1				
S2	-1	-1					2			
S3			-1					1		
S4				-1	0				1	
S5				-1	-1					2

Fig. 7.7. Command Design Pattern Laplacian Matrix – Its five Structors (S1, …, S5) and five Functionals (F1, …, F5) consist together the ten vertices of the bipartite graph (in Fig. 7.6), therefore the Laplacian matrix has 10 columns and 10 rows. Both columns and rows are labelled by the whole vertex set of index labels in the same order. The diagonal (orange) Degree Matrix shows all vertex degrees. The three software modules (blue) in the upper-right quadrant, and reflected in the lower-left quadrant, are the Adjacency Matrix elements showing the neighboring vertices, with a negative sign, by Eq. (7.1).

The Spectral Verification Algorithm is shown in the next text-box.

Spectral Verification Algorithm of Bipartite Graph

Loop

While (Modules-Sparsity > Maximal-Sparsity-Threshold) do

{**Obtain Modules**

Obtain Laplacian Matrix from Bipartite Graph;

Calculate Laplacian eigenvectors/eigenvalues;

Get eigenvectors with 0-valued eigenvalues;

Obtain Modules from these eigenvectors;

Check Sparsity & Obtain Outliers

Calculate Modules-Sparsity;

If (Modules-Sparsity > Threshold)

{***Split Module*** using the Fiedler Vector:

erase edges of Bipartite-Graph;

Potential ***outliers*** = list of erased edges.**}}**

Possible results of the spectral verification algorithm are:

- No outliers = successful design;
- One or more existent outliers = design refactoring is demanded; automatic refactoring is possible for some outlier cases (e.g. [15], [44]).

For a concrete application of the spectral verification to the Command design pattern running example, please look at Sec. 7.5.3.

7.3.5 *Closing the Design Cycle: Modularized Class Diagram*

All the transitions between the algebraic entities described in Fig. 7.3 and in Sec. 7.3, are bijective. For instance, one can pass from the Modularity Matrix to the corresponding Laplacian Matrix, through the intermediate bipartite graph, preserving information concerning Structors, Functionals and Modules. The same is true in the opposite direction, from the Laplacian Matrix to the Modularity Matrix through the intermediate bipartite graph (see e.g. [14]).

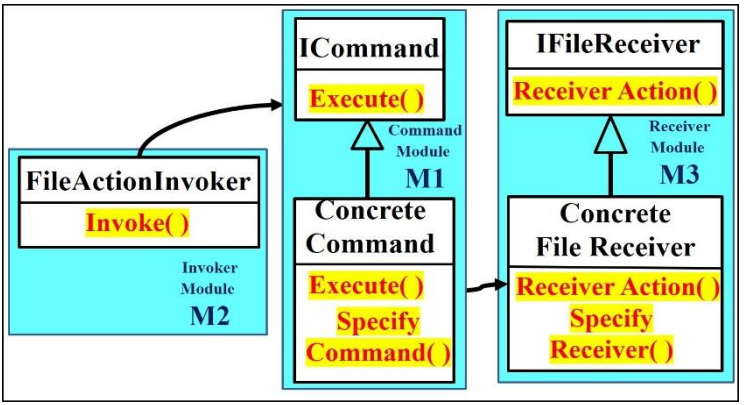


Fig. 7.8. Command Design Pattern Modularized UML Class Diagram — The same classes and interfaces as in Fig. 7.4, with added Modules M1, M2 and M3 (light blue background), as obtained from the Laplacian Matrix eigenvalues and eigenvectors. The modules are the same as in the above matrices, and also in the bipartite graph in Fig. 7.6.

Since, in our running example, all the concepts of the Modularity Matrix were obtained from an UML Class Diagram, Structors and Functionals' information preservation is also extensible from the algebraic entities back to the source Class Diagram. We just add the obtained modules from the Laplacian Matrix, as can be seen in Fig. 7.8. Thus, we have closed the design cycle with a modularized class diagram, showing, the applicability of Linear Software Models, to software design in practice.

7.4 Flexibility of Graphs' Deep Learning

This section ***focuses on graphs***, or equivalent matrices, as the essential Software Design representation in this work. A graph is a flexible structure that can be adapted to any software architecture. We concisely describe GCN (Graph Convolutional Networks) and GAE (Graph Auto-Encoders).

The Nature of Graphs

Graphs may appear as a basic structure inherent to a problem being solved, such as representation and analysis of molecular structures in chemistry. In more interesting situations, graphs are not inherent to the

problem, but are implicit and naturally appear as a medium to achieve a deeper understanding of complex interactions over a latent graph.

An example is dynamic inference of interacting systems, within a team of basketball players and body movements of each player (Kipf *et al.*, 2018, [29]). Another example is complex images understanding, by mapping them to scene graphs of their interrelated objects (Herzig *et al.*, 2018, [25]).

7.4.1 *Graph Convolutional Networks*

Traditional Deep Learning (e.g. [36]) was designed for simple grids in Convolutional Neural Networks (CNN, [48]) or sequences (RNN [40], cf. Sec. 7.2 above). Graph nodes have flexible connections: various neighbor numbers, complex topology and no fixed node ordering. Graph Neural Networks (GNN) operate directly on *graph-structured data.*

GCN (Graph Convolutional Networks) is a type of GNN that takes advantage of a graph dataset by considering adjacent node neighbors in a convolutional fashion. Spectral GCN works in a frequency domain in analogy to Fourier transforms. Spectral filtering, similar to software modularization by Laplacian matrix eigenvectors, succeeded in various tasks, such as graphs classification (Bruna *et al.*, [3], Defferrard *et al.*, [8]).

Spatial GCN is less computationally intensive, working in a spatial domain. It reduces computational costs, while increasing results precision, by simplified operations with restricted radius of local messages among graph nodes and close neighbors. Kipf *et al.* (2018, [29]), define a single node-to-node message passing operation, and operations for moving between node and edge GNN representations.

7.4.2 *Graph Auto-Encoders*

Graph Auto-Encoders (GAE) are the deep learning network architecture currently adopted by this work. GAEs are end-to-end trainable neural network models, used for either unsupervised or semi-supervised learning.

In general, GAE is given a graph adjacency matrix as input and is projected into some latent vectors that capture the graph features, such as the node degree. From this embedded information the GAE reconstructs a

new adjacency matrix with predicted links, that were not present in the inputs. The GAE building blocks are a graph encoder and a decoder.

Here we concisely describe GAEs, based upon the work on Graph Convolutional Matrix Completion by van den Berg, Kipf and Welling [45] (referred in short as BKW). This GAE choice is justified by its network architecture with three basic similarities to our work:

- ***Adjacency Matrix*** — used as input by BKW and also in our software design problem. The Modularity Matrix is in fact an Adjacency matrix, from which the software system bipartite graph is extracted. This is easily perceived in the definition and structure of the Laplacian matrix;
- ***Bipartite Graphs*** — used by BKW in their matrix completion task and also for our software system design (explained in Sec. 7.3);
- ***Link Prediction*** — is part of the BKW matrix completion task and in our design problem, seen as highlighting and eliminating outliers — spurious links disturbing the modularity.

The specific problems solved by BKW — matrix completion by edge prediction — indeed are different from ours — outlier elimination by superfluous edge prediction — but our software design problem has been cast in the same framework as BKW.

In the next paragraphs, we describe the GAE framework in terms of our software design problem, shown schematically in Fig. 7.9.

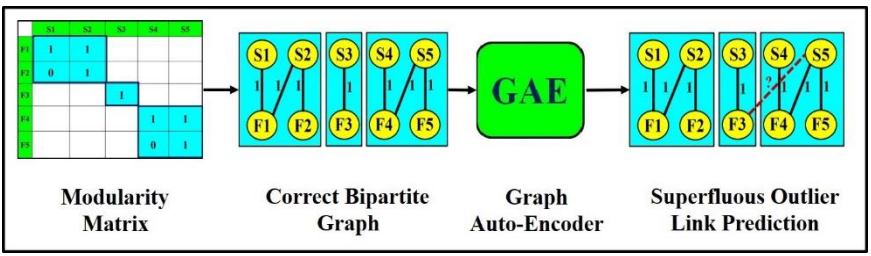


Fig. 7.9. Schematic Graph Auto-Encoder (GAE) Framework for Software System Design — Bipartite Graph, an input to the GAE, is extracted from the Modularity Matrix. In this figure the Modularity Matrix and its bipartite graph are the correct ones, used in the training phase. The GAE output is predictions of superfluous outliers links (see text for additional details).

Three features of the GAE deserve special consideration: (a) the Graph Convolutional Layer; (b) the Decoder; (c) Model Training.

Graphs do not have a lattice-like regular structure, preventing use of common convolutional networks. A **Graph Convolutional Layer** performs only *local operations* that refer to *first-order neighbors of each node*, avoiding expensive and complex topology variability that could occur for more remote neighbors. The same transformation is applied across all graph locations.

Denoting by $\boldsymbol{N_i}$ the set of neighbors of node $\boldsymbol{i}$ and by $\boldsymbol{\mu_{j\to i,w}}$ the message transferred from node $\boldsymbol{j}$ to node $\boldsymbol{i}$, with an edge-weight $\boldsymbol{w}$, messages are accumulated in each node $\boldsymbol{i}$ according to a graph convolution layer, viz. an equation of the form:

$$msgs_i = \sigma\left[\sum_w \left(\sum_{j\varepsilon Ni,1} \mu_{j\to i,1}, \ldots, \sum_{j\varepsilon Ni,W} \mu_{j\to i,W}\right)\right] \tag{7.2}$$

$msgs_i$ is a single hidden parameter vector, resulting from the aggregation by the sum over vectors within parentheses. σ(.) denotes an activation function such as ReLU(.) i.e. Rectified Linear Unit. This graph convolution layer may be followed by one or more layers.

The encoder accumulates information from nodes in the input bipartite graphs. The **Decoder** estimates a probability distribution predicting the weights of superfluous outlier edges in the output bipartite graph (see Fig. 7.9 to the right of the GAE).

Model Training is done by optimization of a cost function (log likelihood or an entropy expression) which considers only the known positive weights in the bipartite graphs to the left of the GAE in Fig. 7.9.

7.5 Software Fusion for Software Design Graphs: Case Studies

We report here two case studies[a] concerning Software Fusion applied to Software Design represented by Bipartite Graphs.

The first case study, with synthetic data, refers to probabilistic Deep Learning of Software Design Graphs. This case study had a double purpose: (a) to check that software design graphs are *reasonably labelled* by the graph classifier; (b) to check that the labelling behavior of the graph

[a]Software used in the Case Study experiments is available upon request.

classifier has the *expected functional dependence* on the input parameters, in order to assure convergence to desired results.

The second case study, upon the Command Design Pattern, refers to deterministic Laplacian verification of graphs obtained by the Deep Learning tool. The purpose of this case study is to show, beyond theoretical correctness ([7], [17]), that Laplacian spectral verification is *applicable in practice*, to software system design. It consisted of a variety of sub-systems, with and without outliers, to test its reliability.

7.5.1 *Graph Deep Learning — Synthetic Data: Experiments*

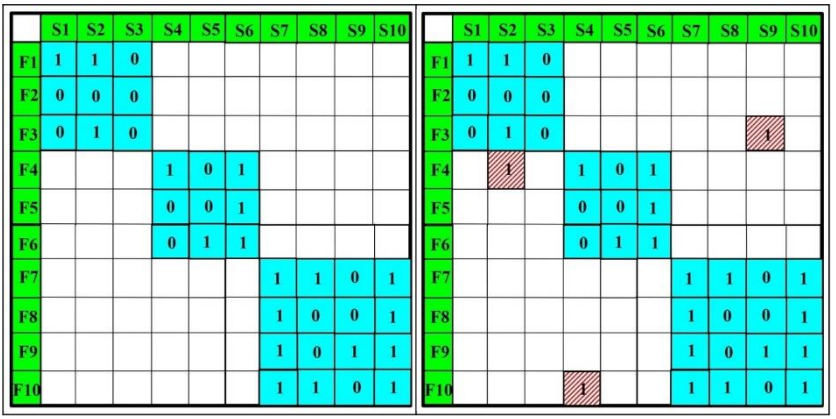
This Case Study evaluates the suitability of Graph Auto-Encoders (GAE) for software design. We trained GAE with pairs of Modularity Matrices: a Modularity Matrix without outliers (M1) and another Matrix with the same overall structure as M1 with added outliers M2 (see Fig. 7.10). A matrix M2 with outliers is the input to the decoder and the original matrix M1 provides the labels for the decoder. Our expectation was that the outliers would be captured and predicted by the GAE network.

The accuracy of predictions was calculated by the number of correctly predicted edges divided by the Total number of edges averaged over the whole test-set by the equation:

$$Accuracy = \frac{Correctly\ Predicted\ Edges}{Total\ Average\ Edges} \tag{7.3}$$

We performed several experiments to evaluate our network with graphs with varying dataset properties as follows:

- ***Graphs count*** — number of graphs in a given test-set;
- ***Modules count*** — how many "modules" the graph contains;
- ***Modules edges sparsity*** — the edges sparsity within the modules;
- ***Nodes count in each vertex set*** — how many Structors and Functionals per bipartite graph;
- ***Outliers sparsity*** — outlier edges sparsity in the whole matrix.



	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
F1	1	1	0							
F2	0	0	0							
F3	0	1	0							
F4				1	0	1				
F5				0	0	1				
F6				0	1	1				
F7							1	1	0	1
F8							1	0	0	1
F9							1	0	1	1
F10							1	1	0	1

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
F1	1	1	0							
F2	0	0	0							
F3	0	1	0						1	
F4		1		1	0	1				
F5				0	0	1				
F6				0	1	1				
F7							1	1	0	1
F8							1	0	0	1
F9							1	0	1	1
F10				1			1	1	0	1

Fig. 7.10. Schematic Sample Modularity Matrices for Experiments — Both matrices have 10 Structors (S1, …, S10) and 10 Functionals (F1, …, F10). The left-hand-side **M1** matrix has no outliers, while the right-hand-side matrix **M2** has 3 outliers (hatched red background). Outliers couple module pairs: for instance the (F4, S2) outlier couples the upper-left and middle modules. Each 1-valued matrix element corresponds to one edge in the respective Bipartite Graphs. Superfluous link prediction identifies outlier edges.

In these experiments, modules were not set by a formal procedure using Laplacian eigenvalues. They were initialized as synthetic randomized sets of neighbor matrix elements. Figure 7.10 shows a ten-node vertex sets example. Our experience with software algebraic representations guided on how to generate synthetic data. Compare Fig. 7.10 with standard Modularity Matrix modules, e.g. in the Command pattern in Fig. 7.5.

Further details on the data used in the experiments are provided next.

Generic Data

Experiments were conducted with a two-layer graph convolution encoder model with 32 and 16 hidden units in each layer. The learning rate was set at 0.01. We used neither weight decay nor dropout.

Typical experiment sizes consisted of a 5 to 1 ratio of training to testing graphs. Each graph has up to 40 nodes in each vertex set of the bipartite graph representing a maximum of 40 Structors and 40 Functionals.

Specific Data

In Experiment #1 specific additional data include: Module density 50%, Graphs count 25, Nodes count 40, and modules count 6.

In Experiment #2 specific additional data were: Modules count 5; Maximal module size 5, Module density 50%, and outliers' density 20%.

In Experiment #3 the minimal module count was 2 as outliers were assumed in this experiment to couple two modules; a single module would not allow outliers existence. Specific additional data were: Module density 50%, Graphs count 25, Nodes count 40, and maximal module size 6.

7.5.2 *Graph Deep Learning — Synthetic Data: Results*

This section shows representative result samples for three experiments.

Experiment #1 refers to accuracy calculations for ***Outliers density vs. Module size*** variations. The GAE prediction accuracy was calculated for varying outlier densities, ranging from 10% to 50% and module sizes ranging from 1 to 6.

Accuracy results are shown in Table 7.1 and also seen in Fig. 7.11. One perceives that larger module sizes are in general more resilient to the presence of outliers (maximal accuracy = 0.60, for outlier density = 0.1, and Module size = 6) since there are enough samples in each module to enable reliable differentiating between different types of graph edges.

Results confirm the classifier expected parameters dependence:

- Accuracy decreases with increasing outlier density and for decreasing module size;
- Larger accuracy dispersion for higher Outliers Density (cf. trendline fittings in Fig. 7.11);

Results fit real software systems' sparse overall matrices, and dense "cohesive" modules, with few outliers relative to module sizes.

Table 7.1 Experiment #1
Accuracy vs. Module Sizes

Module Size	**Accuracy**	
	Outliers' Density = **0.1**	Outliers' Density = **0.5**
1	0.42	0.22
2	0.40	0.34
3	0.49	0.41
4	0.57	0.42
5	0.53	0.37
6	0.60	0.48

[Note: *Outliers Density* = 1 — *Outliers Sparsity*]

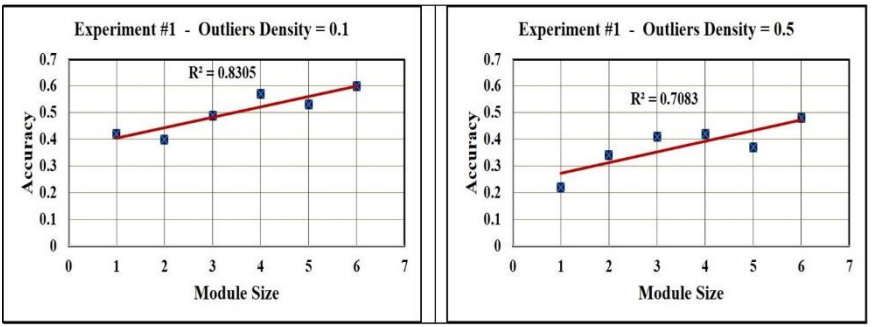


Fig. 7.11. Accuracy vs. Module Sizes for given Outliers' Densities — The left-hand-side chart shows calculations for lower Outliers Density (equals 10%). The right-hand side chart shows calculations for higher Outliers Density (equals 50%). For lower Outliers Density one observes two expected results: (a) the calculated accuracies are generally higher than those with higher Outliers Density; (b) the calculated accuracies have less dispersion for lower Outliers Density, as can be seen from the better fitting to the linear trendline.

In the 2nd experiment, seen in Table 7.2 and in Fig. 7.12, accuracy is calculated as a function of ***Nodes count vs. Graphs count***. The maximal nodes count in each vertex set of the bipartite graph is 40 and the graphs count — ranges from 1 to 75 graphs. The clear result for this experiment is:

- Prediction accuracy increases with nodes count, rather independently of the graphs count used for input, as expected.

Nodes Count is dominant, as graphs count only influences the results averaging. Prediction accuracy improves, as there are more nodes per module from which the network can learn, for constant modules count.

Table 7.2 Experiment #2
Accuracy vs. Nodes Count

Nodes Count	**Accuracy**	
	Graphs Count = **1**	Graphs Count = **75**
8	0.31	0.28
16	0.32	0.39
32	0.45	0.55
40	0.59	0.54

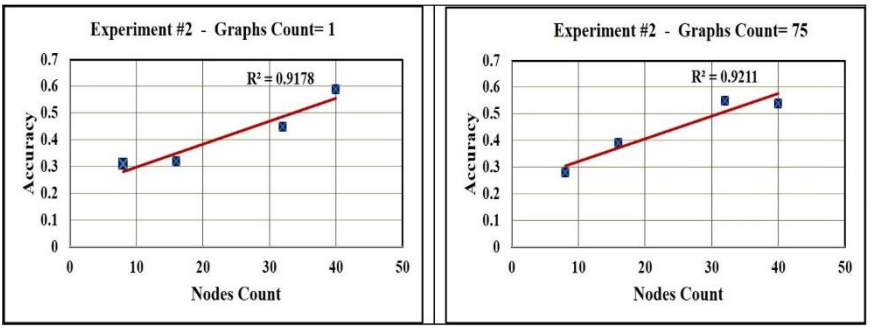


Fig. 7.12. Accuracy vs. Nodes Count for given Graphs Count — The left-hand-side chart shows calculations for Graphs Count = 1. The right-hand-side chart shows calculations for Graphs Count = 75. From both charts one observes the expected result: the calculated accuracy depends only on the Nodes Count, and not on the Graphs Count, as can be seen from the almost identical increasing slopes and the almost identical fittings to the the linear trendlines.

Experiment #3 refers to calculations for ***Outliers density vs. Modules count*** variations. The GAE prediction accuracy was calculated for varying outlier densities, ranging from 10% to 50% and module count from 2 to 6.

Results accuracy can be seen in Table 7.3 and in Fig. 7.13. At first sight one would expect results very similar to the 1st experiment, shown in Fig. 7.11: there one finds (highest accuracy = 0.60, for outlier density = 0.1, and Module size = 6), and here in Fig. 7.13, indeed one has (highest accuracy = 0.62, outlier density = 0.1 and Modules count = 6) — for the same maximal module size = 6.

Experiment #3 is the most surprising and thought provoking of this series. Contrary to expected functionality, the charts in Fig. 7.13 show a big dispersal for outlier density = 0.1 and a decreasing trendline slope for outlier density = 0.3. Two possible interpretations are:

- The probabilistic Deep Learning incorrect graphs labelling reinforces the need for the Deterministic Laplacian matrix Spectral verification and subsequent label correction, eventually leading to *active learning*;
- Since computations have the well-known capability to surprise us, it could be that the seemingly incorrect labelling, is in fact correct, and corresponds to unsuspected novel design graphs of interest, that also would lead to *active learning*, but now in a positive sense.

Table 7.3 Experiment #3
Accuracy vs. Modules Count

Modules Count	**Accuracy**	
	Outliers' Density = **0.1**	Outliers' Density = **0.3**
2	0.41	0.59
3	0.59	0.56
4	0.55	0.53
5	0.50	0.52
6	0.62	0.52

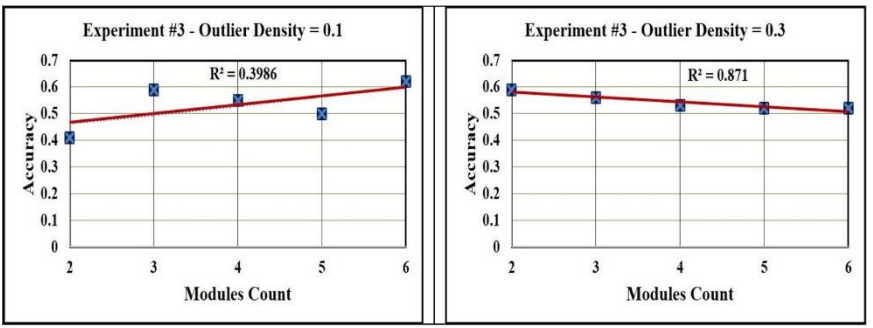


Fig. 7.13. Accuracy vs. Modules Count for given Outliers' Densities — The left-hand-side chart shows results for lower Outliers Density (equals 10%); one observes a very high dispersion of the calculated values. The right-hand-side chart shows results for higher Outliers Density (equals 30%); besides the surprisingly high accuracy range for a higher outliers' density, the decreasing slope of the linear trendline is the opposite of the expected functionality. See text for two possible interpretations.

Summarizing the experiment results:

- In Experiments #1 and #2 results are rather consistent with expectations for the calculated accuracies, as a function of the relevant parameters.
- In Experiment #3 results reinforce the Software Fusion approach, viz. the need for the Deterministic Laplacian matrix Spectral verification and subsequent label correction of the labelled graphs.

7.5.3 *Command Design Pattern — Laplacian Spectral Verification: Experiments*

This section describes the second Case Study, viz. deterministic Laplacian Spectral Verification applied to the often used Command Design Pattern (see its description in the Design Patterns GoF book [18], pp. 233–242).

The experiments consisted in:

- Laplacian Matrix Generation from input bipartite graphs representing specific software designs;
- Laplacian eigenvectors' calculation, focusing on modules and Fiedler vectors which suggest splitting modules in the presence of outliers.

Figure 7.14 shows four Modularity Matrices used in the experiments:

(I) **Mod0** — Command Design Pattern without outliers;
(II) **Mod1** — Command pattern with one added distant outlier;
(III) **Mod2** — Command pattern with one outlier adjacent to a module;
(IV) **Mod3** — Command pattern with two outliers.

Modularity Matrices are convenient to visualize the different situations. One easily extracts the fitting bipartite graph and derives the Laplacian Matrix from each Modularity Matrix as detailed in Sec. 7.3.

7.5.4 *Command Design Pattern —Laplacian Spectral Verification: Results*

Figure 7.14 shows Modularity Matrices, while the results in Fig. 7.15 refer to their Laplacian matrices. Modularity Matrices have size 5*5, and the respective Laplacians have size 10*10. Their eigenvectors in Fig. 7.15 have 10 vector elements.

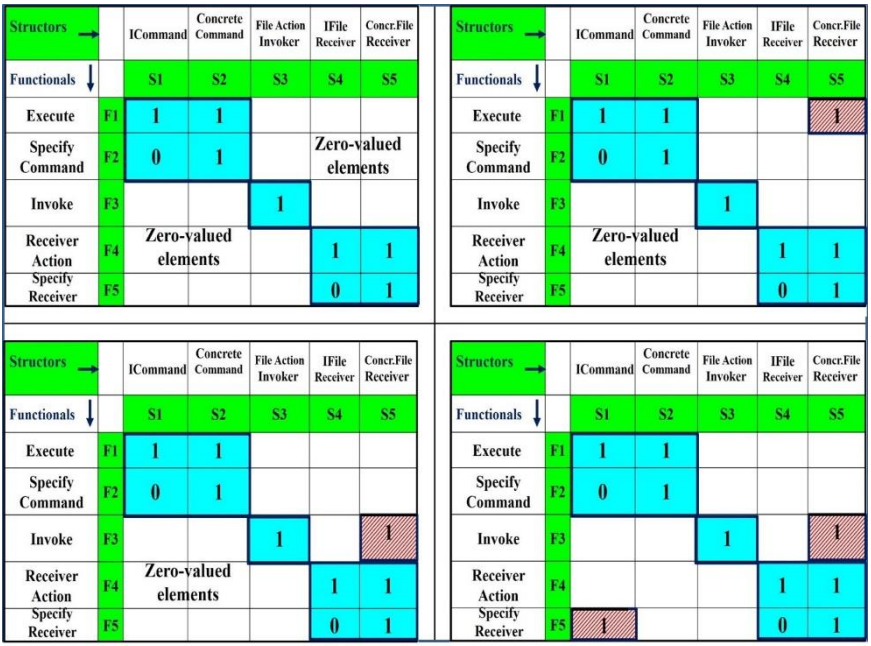


Fig. 7.14. Command Design Pattern: four Modularity Matrices and outliers — The upper-left Modularity Matrix **Mod0** has three (blue) modules and no outliers. The upper-right Matrix **Mod1** has one added outlier (hatched red background) in an element (***F1***, ***S5***) distant from any module. The lower-left Matrix **Mod2** has one outlier in an element (***F3***, ***S5***) adjacent to the lower-right module. The lower-right Matrix **Mod3** has two outliers.

Matrix results are shown in Fig. 7.15. Matrix **Mod0** has 3 low sparsity modules (see Fig. 7.14) and no outlier. There is no need to split modules, thus the Fiedler eigenvector is *not necessary*.

Matrix **Mod1** has only two modules: the original small 1*1 middle module, and a big module of size 4*4 in which the distant outlier, in Matrix element (***F1***, ***S5***) couples the two (upper-left and lower-right) modules of size 2*2 of the original matrix **Mod0**. As the coupled 4*4 module is sparse, it is split by the signs of the Fiedler vector elements. The Fiedler vector (in Fig. 7.15), negative elements (**F1**, **F2**, **S1**, **S2**) form one split module and positive elements (**F4**, **F5**, **S4**, **S5**) form the other split module. Two zero-valued vector elements (**F3**, **S3**) fit the uncoupled 1*1 middle module, not appearing in the Fiedler vector, as it splits only the sparse coupled module.

Matrix	Modules	Outliers	Laplacian Module Eigenvectors	Laplacian Fiedler Eigenvalues / Eigenvectors
Mod0	**3**	**none**	**((1,1,0,0,0,1,1,0,0,0), (0,0,1,0,0,0,0,1,0,0), (0,0,0,1,1,0,0,0,1,1))**	**Not necessary**
Mod1	**2**	**1 distant**	**F1 F2 F4 F5 S1S2 S4 S5 ((1,1,0,1,1,1,1,0,1,1), (0,0,1,0,0,0,0,1,0,0)) F3 S3**	**Eigenvalue = 0.251 F1 F2 F3 F4 F5 (-0.162, -0.523, 0.000, 0.392, 0.217, -0.217, -0.392, 0.000, 0.523, 0.162) S1 S2 S3 S4 S5**
Mod2	**2**	**1 adjacent**	**((1,1,0,0,0,1,1,0,0,0) (0,0,1,1,1,0,0,1,1,1))**	**Not necessary (see text)**
Mod3	**1**	**two**	**(1,1,1,1,1,1,1,1,1,1)**	**Eigenvalue = 0.128 F1 F2 F3 F4 F5 (0.323, 0.511, -0.282, -0.282, -0.024, 0.160, 0.445, -0.323, -0.323 -0.205) S1 S2 S3 S4 S5**

Fig. 7.15. Command Design Pattern: Laplacian Modules and Fiedler eigenvectors — These are calculation results of the four matrices of Fig. 7.14. It shows modules and outlier numbers. The respective Laplacian module eigenvectors highlight their 1-valued elements (yellow background). The Fiedler eigenvectors have marked positive elements (blue background) and negative elements (green background). Eigenvector elements are arranged by the index order of the Laplacian: F1, F2, F3, F4, F5, S1, S2, S3, S4, S5.

Concerning Matrix **Mod2**, Fig. 7.14 visually suggests that there is an outlier element (***F3***, ***S5***) adjacent to the lower-right 2*2 module. However, the Laplacian spectral verification algorithm, correctly decides that the element (***F3***, ***S5***) is not an outlier, but in *design terms* it is an element of a legitimate 3*3 *non-sparse* (4 zero-valued out of 9 matrix elements) and block-diagonal module. Thus, the Fiedler eigenvector is not necessary.

This is a most interesting case, since the adjacent outlier seems to imply that the **Mod2** design has been generated by different requirements than **Mod0**. The Laplacian Verifier correctly pinpoints this discrepancy. Therefore, only a human software engineer can decide whether in *conceptual terms* this 3*3 module fits reasonable software requirements from which such a design could be obtained.

Matrix **Mod3** apparently has just one module due to two coupling outliers, 1 distant outlier and another adjacent outlier of the types seen in the two previous matrices. Again, the Laplacian spectral algorithm

correctly performs its verification role. The Fiedler vector splits the single big module of size 5*5, into two smaller modules of 2*2 and 3*3 sizes and stops there. The 3*3 module is legitimate by the same reason as in **Mod2**.

Summarizing these experiments:

- The Laplacian spectral algorithm succeeded in various tests, including boundary situations; thus, it is applicable as a deterministic *design verifier* to assert correctness of a probabilistic deep learning classifier.

7.6 Related Work

The literature on Deep Learning applications to software is extensive. Here we concisely review relevant topics. Active Learning is used within Software Fusion (in Fig. 7.1). Other issues are alternative approaches to Graphs Deep Learning, and Theoretical Principles for network architectures. Finally, this short review delineates the scope of this work.

7.6.1 *Active Learning*

Active Learning is an iterative procedure to select the most informative examples of a subset of unlabeled — or mislabeled — samples. The choice is based upon scores from a given model outcome. The chosen candidates are added to the training set, and the classifier is retrained. P. Liu *et al.* (2016, [35]) applied active learning to classify hyperspectral images, since it is expensive to get well labeled samples for remote sensing applications.

Another interesting Active Learning application is the paper by Stark *et al.* [43] using Deep Learning for CAPTCHA recognition. CAPTCHAs are automated tests to distinguish humans from robots, i.e. easily solvable by humans, but not by robots. But! Convolutional Neural Networks (CNN) do solve these tests. The problem is large amounts of training data. Active Deep Learning is a strategy to overcome the training data problem.

7.6.2 *Graphs Deep Learning Alternatives*

GraphSAGE, introduced by Hamilton, Ying and Leskovec, in their 2017 paper “Inductive Representation Learning on Large Graphs” [24], is a framework with *inductive* capability to generate node embeddings for

previously unseen data. It learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood.

GIN (Graph Isomorphism Network) is a neural architecture by Xu *et al.* (2019, [49]), motivated by a theoretical approach (cf. the next section), is claimed to be the most expressive among a class of GNNs. It works by an aggregation function capturing the full multiset of node neighbors.

Chen *et al.* (2020, [4]) propose a Linear Residual GCN approach. It is relevant reading for a few reasons. (a) They claim that removing non-linearities enhances performance. (b) They treat user-item historical behavior as a bipartite graph. (c) It discusses the issue that in practice, most GCN based variants achieve the best performance with just 2 layers.

7.6.3 *Theoretical Approaches for Neural Network Architectures*

Various approaches tried to formalize Deep Learning Neural Networks (DNN), to understand DNN's inner workings and their limitations. We mention four approaches: (a) probabilistic representations; (b) model reformulation; (c) theoretical principles; (d) formal bounds.

An early example of ***Probabilistic Representations*** is a paper by Patel *et al.* (2015, [39]), proposing a Probabilistic Theory of Deep Learning. They contrast Deep Learning success with remaining fundamental questions: Why do they work? Understanding and synthesizing deep learning architectures have remained elusive. They propose a framework based upon a Bayesian generative probabilistic model. A recent example is the paper by Lan and Barner (2019, [32]). They offer an energy-based model that can also be interpreted as a Bayesian neural network.

Model Reformulation translates a set of equations of a model, into another set of equations recognizable as a different but known model, with familiar properties. This enables re-interpretation of the initial model.

A paper by Li, Han and Wu (2018, [33]), obtains deeper insights into GCN for semi-supervised learning, showing that the graph convolution within GCN is a special form of Laplacian smoothing — symmetric Laplacian smoothing. Laplacian smoothing calculates new vertex features as the weighted average of itself and its neighbors'. Vertices in the same cluster tend to be densely connected, thus smoothing makes their features similar, causing the subsequent classification task to be much easier.

Theoretical principles seek deeper understanding of the DNN's inner workings for specific classes of neural networks architectures. A recent example from the literature on GCNs, is a design principle that follows from a natural requirement of "*permutation invariance*". See e.g. the paper by Zaheer *et al.* (2017, [51]) and the paper by Herzig *et al.* (2018, [25]).

Explicit ***Formal Bounds*** try to answer questions about Neural Networks limitations. The paper by Xu, Hu, Leskovec and Jegelka, [49], already mentioned in Sec. 6.2 as the proposer of GIN, is significantly entitled "How Powerful are Graph Neural Networks?". It proposes the Weisfeiler-Lehman (WL) graph isomorphism test as a source of a formal upper bound for expressiveness of neural network architectures.

The WL graph isomorphism test [46] was formulated in 1968, outside the Neural Networks context. It is able to distinguish a broad class of graphs. The WL test iteratively updates a given node's feature vector by injective aggregation that maps different node neighborhoods to different feature vectors. Xu *et al.* claim that a GNN can have as large discriminative power as the WL test if the GNN's aggregation scheme can model injective functions. An earlier paper by Shervashidze *et al.* (2011, [42]), entitled "Weisfeiler-Lehman Graph Kernels", used a rapid feature extraction based on the WL test, without formulating a formal bound.

A recent paper by Garg, Jegelka and Jaakkola from February 2020 [19] mentions models that are more powerful than the WL test. Beyond *Representational* limits of GNN, the latter paper provides the first data dependent *Generalization* bounds for message passing GNNs.

7.6.4 *Other Possible Applications*

A recent paper by Zhang *et al.* (2019, [52]) describes DeepCheck, a deep learning technique detecting program code attacks, in the executable file. It acts on the control flow graph of the code to detect abnormal flows.

In order to keep this work consistent and self-contained, we have delineated its boundaries. It was decided that implementation of software system design in programming languages, or run-time behavior issues are out of the scope of this work.

7.7 Discussion

This final Discussion touches foundational issues in this chapter: (a) the importance of Conceptual Integrity preservation: the connection between concepts and algebra; (b) the ultimate meaning of Software Fusion; (c) obstacles to Complete Automation. The Discussion concludes mentioning future work and the Main contribution of this chapter.

7.7.1 *Preserving Conceptual Integrity: Between Software Concepts and Algebra*

Software design within Linear Software Models involves a consistent connection between two apparently unrelated and very different kinds of ideas: ***software concepts*** and ***algebraic structures***. This connection triggers the following question: Why is Conceptual Integrity preservation throughout modularization within Linear Software Models so important?

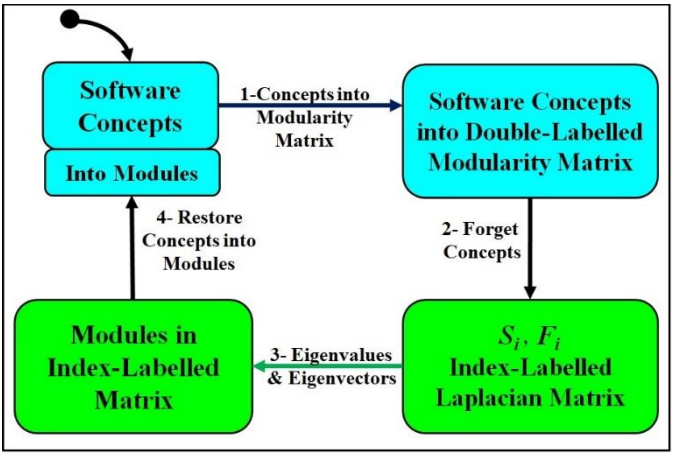


Fig. 7.16. ***Modularization Process: Software Concepts*** & ***Algebraic Structures*** — One starts a software system characterization with Software Concepts. The next four steps are: 1- Insert Concepts into the Double-Labelled Modularity Matrix; 2- Forget Concepts; 3- Calculate Laplacian Matrix eigenvalues and eigenvectors obtaining System Modules; 4- Restore Concepts into the software system Modules. From step 1 onwards the Concepts and Algebraic Structures are never disconnected. Initially the double Labels, and throughout the process the short index-Labels, keep track of the Concepts associated to the respective index-Labels.

To answer this question, we first look at the modularization process from a slightly different perspective in Fig. 7.16.

The four steps in Fig. 7.16 are:

1. ***Characterize*** *the* ***software system*** *by* ***software concepts*** — Software system design starts with ***Concepts*** and their relationships chosen according to the *human* software engineer *understanding* of the software system requirements, (cf. beginning of Sec. 7.3.2). One obtains a bipartite graph with vertices doubly identified by the software concepts and in a one-to-one correspondence by unique short index labels $\boldsymbol{S_i}$, $\boldsymbol{F_i}$ (devoid of semantic content), one label per vertex.
2. ***Represent*** *the software system* ***conceptual graph*** *by* ***algebraic structures*** — these are matrices, preserving the whole semantic information from the conceptual graph, in which matrix columns and rows are identified by the unique labels as the graph vertices.
3. ***Forget*** *the software concepts and perform* ***algebraic manipulations*** *on the* ***algebraic structures*** — keep only the unique short index labels, as software concepts have no influence whatsoever on the algebraic manipulations. These calculate matrix eigenvalues and eigenvectors.
4. ***Modularize*** *the* ***software system****, restoring the* ***software concepts partitioned into modules*** — Once software concepts are visible, one can double-check the modularization suitability, taking into account semantic considerations. The software engineer may either modify the sets of software concepts or re-assign methods to different classes and restart the four steps of the whole process.

Two desirable properties of these four steps are:

- **Total formal independence** — software concepts and algebraic structures are completely independent:
  in step (1) — the engineer is free to determine the software system semantic content, choosing concepts and assigning methods to classes;
  in step (3) — algebraic manipulations are not affected at all by the conceptual semantic content, contributing to an independent critical view of the software system.
- **Complete preservation of the semantic links along all steps** — assured by bijective double identifiers of graph vertices and matrix columns/rows.

Indeed, ***software concepts*** and ***algebraic structures*** are intimately connected all along the design modularization process, while keeping their rigorous formal independence.

Now we can answer to the question at the beginning of this section.

Conceptual Integrity preservation throughout the modularization process within Linear Software Models is so important for two reasons:

- *Human understanding of the process* — The software engineer never loses understanding of the conceptual content of operations during system modularization.
- *Response to conceptual disconnect between algebra and semantics* — This solves the *conceptual disconnect* between algebra and semantics, occurring in lower-abstractions learning, pointed out in Sec. 7.2.

7.7.2 *The Ultimate Meaning of Software Fusion*

Software Fusion, the approach described in this work, has been defined from the beginning of this chapter, in Fig. 7.1, as the fusion of two complementary, but independent, mathematical tools:

- 1st Tool — Probabilistic Deep Learning;
- 2nd Tool — Deterministic Laplacian matrix Spectral Verification of the output of the 1st Tool.

The ultimate Software Fusion, could in principle, merge these independent tools into a single unified tool. Indeed, the GCN architecture by Defferrard *et al.* described in the paper from 2016 [8] integrates spectral filtering by Laplacian eigenvectors within Deep Learning.

But merging the two tools has three disadvantages:

- Defferrard *et al.* state that the spectral filtering by Laplacian eigenvectors is very costly in terms of multiplication operations;
- Spectral methods have a serious drawback, compared to neighborhood aggregation within spatial approaches: the Laplacian graph needs to be known in advance. They cannot generalize to unseen graphs (cf. [49]).
- The deterministic spectral verification of the results would be lost since it is embedded inside the probabilistic Deep Learning.

The current reply to the first disadvantage, is the more recent GAE (Graph Auto-Encoder) architecture by Kipf, Welling *et al.* found in the papers from 2016 [30] and from 2017 [31]. It is more efficient, by means of reasonable approximations, using the message passing approach, instead of direct use of the Laplacian eigenvectors within the network.

A possible reply to the third disadvantage, poses the next question:

- Can one have a Conceptual Integrity preservation throughout the Graph Deep Learning process similar to the Conceptual Integrity preservation throughout the modularization process within Linear Software Models?

A cautious reply to this question would be that it depends on the Deep Learning model. According to the paper by Li, Han and Wu [33] (referred to in Sec. 7.6.3) GCN graph convolution is a special form of Laplacian smoothing, calculating vertex features as the weighted average of itself and its neighbors'. Since vertices in the same cluster tend to be densely connected — which is reasonable for cohesive modules — smoothing makes their features similar, blurring vertex identities, causing loss of Conceptual Integrity preservation. This issue needs further investigation.

7.7.3 *Possible Obstacles to Complete Automation*

We aim, in the long term, at automatic generation of Software System Design. One can identify two possible obstacles to complete automation:

- *Stopping criterion of the Laplacian modularization loop* — the Spectral Verification Algorithm of the Bipartite Software Design Graph in Sec. 7.3.4 has a loop with a criterion to stop splitting a too sparse module. The criterion is set by a module sparsity threshold. There is no danger of a non-terminating loop. A possible problem is stopping either too early or too late. This issue should be further investigated;
- *Criteria to label output graphs of the Laplacian spectral verification* — in Fig. 1, the output of the Deterministic Laplacian spectral verification, either has the correct label and is added to the final results, or has an incorrect label which must be corrected, and eventually reused for Active Learning, i.e. re-training the Deep Learning GCN. The decision whether reuse for Active Learning still must be precisely formulated.

7.7.4 *Future Work*

Besides previous sections' open issues, future work planned items are:

- Extending bipartite graphs with provider only structors, to bipartite graphs with provider and consumer structors, as a more complete algebraic description of software systems [16];
- Since Graphs Deep Learning is a very dynamic research field, we shall consider emerging alternative models for software system design.

7.7.5 *Main Contribution*

The main contribution of this work is the *Software Fusion* of ***Probabilistic*** Deep Learning with ***Deterministic*** Laplacian Matrix spectral verification in order to assure correctness of Software System Design.

This involved the following three cardinal decisions:

- Work in the higher-abstraction level of *Software Design*;
- Represent software design by *Bipartite Graphs*;
- Apply Deep Learning by *Graph Convolutional Neural Networks*.

References

1. Brooks, F. P. (1995). *The Mythical Man-Month — Essays in Software Engineering* — Anniversary Edition, Addison-Wesley, Boston, MA, USA.
2. Brooks, F. P. (2010). *The Design of Design: Essays from a Computer Scientist*, Addison-Wesley, Boston, MA, USA,
3. Bruna, J., Zaremba, W., Szlam, A. and LeCun, Y. (2014). Spectral Networks and Locally Connected Networks on Graphs. https://arxiv.org/abs/1312.6203 [cs.LG].
4. Chen, L., Wu, L., Hong, R., Zhang, K. and Wang, M. (2020). Revisiting Graph Based Collaborative Filtering: A Linear Residual Graph Convolutional Network Approach, *in Proc. 34th AAAI Conf.* pp. 27–34.
5. Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. arXiv:1406.1078.
6. Dam, H. K., Tran, T., Grundy, J. and Ghose, A. (2016). DeepSoft: A Vision for a Deep Model of Software. https://arxiv.org/abs/1608.00092 [cs.SE].

7. De Abreu, N. M. M. (2007). Old and new results on algebraic connectivity of graphs, Linear Algebra and its Applications, 423, pp. 53–73. DOI: 10.1016/j.laa.2006.08.017.
8. Defferrard, M., Bresson, X. and Vandergheynst, P. (2016) Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, in Proc. 30th Conf. Neural Information Processing Systems — NIPS, Barcelona, Spain.
9. De Rosso, S.P. and Jackson, D. (2013). What's Wrong with Git? A Conceptual Design Analysis, in *Proc. of Onward! Conference*, pp. 37–51, ACM. DOI: 10.1145/2509578.2509584.
10. Exman, I. (2012). Linear Software Models, Extended Abstract, in Jacobson, I., Goedicke, M. and Johnson, P. (eds.), *GTSE 2012, SEMAT Workshop on General Theory of Software Engineering*, pp. 23–24. KTH Royal Institute of Technology, Stockholm, Sweden. Video: http://www.youtube.com/watch?v=EJfzArH8-ls.
11. Exman, I. (2014). Linear Software Models: Standard Modularity Highlights Residual Coupling, Int. Journal on Software Engineering and Knowledge Engineering, vol. 24, pp. 183–210. DOI: 10.1142/S0218194014500089.
12. Exman, I. (2015). Linear Software Models: Decoupled Modules from Modularity Matrix Eigenvectors, Int. Journal on Software Engineering and Knowledge Engineering, vol. 25, pp. 1395–1426. DOI: 10.1142/S0218194015500308.
13. Exman, I. and Katz, P. (2016). Modulaser: A Tool for Conceptual Analysis of Software Systems, in *Proc. SKY'2016 7th Int. Workshop on Software Knowledge*, pp. 19–26. DOI: 10.5220/0006080700190026.
14. Exman, I. and Sakhnini, R. (2018). Linear Software Models: Bipartite Isomorphism between Laplacian Eigenvectors and Modularity Matrix Eigenvectors, Int. Journal of Software Engineering and Knowledge Engineering, Vol. 28, No 7, pp. 897–935. DOI: 10.1142/S0218194018400107.
15. Exman, I. and Nechaev, A. (2020). Algebraic Higher-Abstraction for Software Refactoring Automation, in *Proc. SEKE'2020 Int. Conf. on Software Engineering and Knowledge Engineering*. DOI: 10.18293/SEKE2020-008.
16. Exman, I. and Wallach, H. (2020). Linear Software Models: An Occam's Razor Set of Algebraic Connectors Integrates Modules into a Whole Software System, Int. Journal of Software Engineering and Knowledge Engineering, Vol. 30, No 10, pp. 1375–1413. DOI: 10.1142/S0218194020400185.
17. Fiedler, M. (1973). Algebraic Connectivity of Graphs, *Czech. Math. J.*, Vol. 23, (2) 298–305.

18. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, USA.
19. Garg, V. K., Jegelka, S. and Jaakkola, T. (2020). Generalization and Representational Limits of Graph Neural Networks, https://arxiv.org/abs/2002.06157 [cs.LG].
20. Goyal, P. and Ferrara, E. (2017). Graph Embedding Techniques, Applications and Performance: A Survey, https://arxiv.org/abs/1705.02801 [cs.SI].
21. Gu, X., Zhang, H., Zhang, D. and Kim, S. (2017). Deep API Learning, Proc. FSE'16, https://arxiv.org/abs/1605.08545 [cs.SE]. DOI: 10.1145/1235.
22. Guo, J., Chang, J. and Cleland-Huang, J. (2018). Semantically Enhanced Software Traceability Using Deep Learning Techniques, https://arxiv.org/abs/1804.02438 [cs.SE].
23. Gupta, R., Pal, S., Kanade, A. and Shevade, S. (2017). DeepFix: Fixing Common C Language Errors by Deep Learning, *Proc. 31st AAAI Conference*, pp. 1345–1351.
24. Hamilton, W. L., Ying, R. and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs, *Proc. 31st NIPS Conf. Info. Processing Systems*. {GraphSAGE}.
25. Herzig, R., Raboh, M., Chechik, G., Berant, J. and Globerson, A. (2018). Mapping Images to Scene Graphs with Permutation-Invariant Structured Prediction, https://arxiv.org/abs/1802.05451 [stat.ML].
26. Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory, *Neural Computation*, 9(8):1735–1780. DOI: 10.1162/neco.1997.9.8.1735 URL: http://www.bioinf.jku.at/publications/older/2604.pdf.
27. Hu, X., Li, G., Xia, X., Lo, D. and Jin, Z. (2018). Deep Code Comment Generation, *Proc. ICPC IEEE/ACM Int. Conf. Program Comprehension*, 11 pages. DOI: 10.475/123_4.
28. Kalofolias, V., Bresson, X., Bronstein, M. and Vandergheynst, P. (2014). Matrix Completion on Graphs, https://arxiv.org/abs/1408.1717 [cs.LG].
29. Kipf, T. N., Fetaya, E., Wang, K-C., Welling, M. and Zemel, R. (2018). Neural Relational Inference for Interacting Systems, https://arxiv.org/abs/1802.04687 [stat.ML].
30. Kipf, T. N. and Welling, M. (2016). Variational Graph Auto-Encoders, in *NIPS Bayesian Deep Learning Workshop*, https://arxiv.org/abs/1611.07308 [stat.ML].
31. Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks, *Proc. ICLR*, https://arxiv.org/abs/1609.02907 [cs.LG].
32. Lan, X. and Barner, K. E. (2019). A Probabilistic Representation of Deep Learning, https://arxiv.org/abs/1908.09772 [cs.LG].

33. Li, Q., Han, Z. and Wu, X-M. (2018). Deeper Insights into Graph Convolutional Networks for Semi-supervised Learning, https://arxiv.org/abs/1801.07606 [cs.LG].
34. Li, X., Jiang, H., Ren, Z., Li, G. and Zhang, J. (2018). Deep Learning in Software Engineering, https://arxiv.org/abs/1805.04825 [cs.SE].
35. Liu, P., Zhang, H. and Eom, K. B. (2016). Active Deep Learning for Classification of Hyperspectral Images, https://arxiv.org/abs/1611.10031.
36. Ma, Y., Jin, W., Tang, J., Wu, L. and Ma, T. (2020). Graph Neural Networks: Models and Applications, AAAI Tutorial, http://cse.msu.edu/~mayao4/tutorials/aaai2020/.
37. Meyes, R., Lu, M., de Puiseau, C. W. and Meisen, T. (2019). Ablation Studies in Artificial Neural Networks, https://arxiv.org/abs/1901.08644 [cs.NE].
38. Mou, L., Men, R., Li, G., Zhang, L. and Jin, Z. (2015). On End-to-End Program Generation from User Intention by Deep Neural Networks, https://arXiv.org/abs/1510.07211 [cs.SE].
39. Patel, A. B., Nguyen, T. and Baraniuk, R. G. (2015) A Probabilistic Theory of Deep Learning, https://arxiv.org/abs/1504.00641 [stat.ML].
40. Rumelhart, D. E., Hinton, G.E. and Williams, R.J. (1986). Learning representations by error propagation, In Rumelhart, D. E., McClelland, J. L. and the PDP Research Group (Eds.), Parallel Distributed Processing (Vol. 1, pp. 318–362). MIT Press, Cambridge, MA; originally in *Nature, Vol. 323 (6088): pp. 533–536 (1986). DOI:* 10.1038/323533a0.
41. Schlichtkrull, M., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I. and Welling, M. (2017). Modeling Relational Data with Graph Convolutional Networks, https://arxiv.org/abs/1703.06103 [stat.ML].
42. Shervashidze, N., Schweitzer, P., van Leewen, E. J., Mehlhorn, K. and Borgwardt, K. M. (2011). Weisfeiler-Lehman Graph Kernels, J. Machine Learning Research, vol. 12, pp. 2359–2562.
43. Stark, F., Hazirbas, C., Triebel, R. and Cremers, D. (2015). CAPTCHA Recognition with Active Deep Learning, Proc. Workshop New Challenges in Neural Computation.
44. Szoke, G., Nagy, C., Ferenc, R. and Gyimothy, T. (2016). Designing and Developing Automated Refactoring Transformations: An Experience Report, 23rd IEEE Int. SANER Conf., Vol. 5, pp. 693–697. DOI: 10.1109/SANER.2016.17.
45. van den Berg, R., Kipf, T. N. and Welling, M. (2017). Graph Convolutional Matrix Completion, https://arxiv.org/abs/1706.02263 [stat.ML].
46. Weisfeiler, B. and Lehman, A. A. (1968). A reduction of a graph to a canonical form and an algebra arising during this reduction, Nauchno-Technicheskaya Informatsia, 2(9) pp. 12–16.
47. Weisstein, E. W. (2020). Bipartite graph http://mathworld.wolfram.com/Bipartite-Graph.html.

48. Wu, J. (2017). Introduction to Convolutional Neural Networks, Nanjing University, China. https://pdfs.semanticscholar.org/450c/a19932fcef1ca6d0442cbf52fec38fb9d1e5.pdf.
49. Xu, K., Hu, W., Leskovec, J. and Jegelka, S. (2019). How Powerful are Graph Neural Networks?, Proc. ICLR 2019, https://arxiv.org/abs/01810.00826 [cs.LG]. {GIN}
50. Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K. and Jegelka, S. (2018) Representation Learning on Graphs with Jumping Knowledge, https://arxiv.org/abs/1806.03536 [cs.LG].
51. Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R. and Smola, A. J. (2017). Deep Sets, in Advances in Neural Information Processing Systems (NIPS).
52. Zhang, J., Chen, W. and Niu, Y. (2019). DeepCheck: A Non-intrusive Control-flow Integrity Checking based on Deep Learning, https://arxiv.org/abs/1905.01858 [cs.CR].

https://doi.org/10.1142/9789811239922_0008

Chapter 8

Using Artificial Intelligence for Auto-Generating Software for Cyber-Physical Applications

Gregory Provan

School of Computer Science and IT, Western Gateway Building, University College Cork, Cork, Ireland T12 WT72
g.provan@cs.ucc.ie

8.1 Introduction

Model-based software engineering (MBSE) is the process of creating and exploiting domain models to generate software that can be verified to satisfy particular requirements R [1]. A model Φ is an abstract representation of the knowledge and activities that govern a particular application domain. The MBSE approach aims to increase productivity by (a) simplifying the design process through using models of recurring design patterns in the application domain; and (b) maximizing compatibility between systems through reuse of these standardized models.

Generating software that guarantees requirements $\mathcal{R}$ is an important task that traditionally has followed a manual process from requirements through to software [1]. For some cyber-physical systems (CPSs) for which models of nominal and faulty performance exist, this manual process can be augmented with the use of pre-defined system models. For example, this can be done for control systems using MATLAB/Simulink model libraries (www.mathworks.com/) or for other systems using languages like Modelica (e.g., www.modelica.org/libraries).

Traditional MBSE uses entirely human-defined artefacts, e.g., physical model components and fault trees, so it produces explainable outputs that meet particular safety requirements. However, a drawback of traditional model-based development is the need to manually generate the model

libraries and to assemble to pre-defined components into complete systems (e.g., [1]). A second drawback is that the models generated from "generic" components must have their parameters tuned to the specific application (e.g., [2]).

To circumvent these drawbacks, artificial intelligence (AI) techniques have been increasingly been proposed for generating software that guarantees we satisfy requirements $\mathcal{R}$ (e.g., [3–5]). These proposals range from using theorem provers throughout the process, replacing the entire process with machine learning [6] to applying learning to specific sub-sets of the process [7]. For example, Moitra *et al.* [8] discuss a tool that enables users to write requirements that are clear, unambiguous, conflict-free and complete. This tool creates requirements in a structured natural language that is both human- and machine-readable, and uses an automated theorem prover to formally verify the requirements and identify errors. Methods that use learning-based automation reduce costs, but may also reduce the explainability and trustworthiness of the generated systems. Automated MBSE systems don't incorporate AI/learning, although there are some recent proposals to do so [9].

At present, little focus is placed on the costs of the different phases of MBSE, and in particular the costs of model-generation. Given a new framework in which we can define different levels of automation in model-generation, it is important to explicitly examine the trade-offs associated with different model construction methods and the resulting models.

This article first reviews the most prominent proposals for applying AI to generating verified software and proposes an approach that integrates learning withing the model-based development process. We compare several different AI-based methods in terms of accuracy, and being explainable and trustworthy. We empirically show how this approach works for a CPS application. We focus on the model-generation phase, where we create a system model that guarantees properties defined with requirements $\mathcal{R}$.

Our contributions are as follows:

- We compare and contrast traditional MBSE with AI-based MBSE to highlight their strengths and weaknesses.
- We introduce a new framework for the model-development phase of MBSE, using an optimization framework to formalize both the traditional and AI approaches.
- We illustrate these differences using a CPS application of a chemical process system with fault-tolerance guarantees.

This article is organized as follows. Section 8.2 reviews the state-of-the-art in MBSE and learning-based model construction. Section 8.3 introduces the area of MBSE. Next, Sec. 8.4 introduces the phases of MBSE, and illustrates them with a well-known process-control example. Section 8.5 describes the new optimization-based framework for MBSE. Section 8.6 describes the technical details of the proposed new framework. Section 8.8 describes the different model-generation methodologies that can be used for semi- or fully-automated model generation within this new framework.

Section 8.9 presents an empirical comparison several examples of the new MBSE process. Section 8.10 summarizes our conclusions.

8.2 Related Work

Researchers have proposed two main methods for performing MBSE tasks:

(1) **Model-Based (formal methods)**: Manually build models, which can be used to formally verify if the requirements are satisfied [2, 10].
(2) **Machine Learning**: Learn representations from data that can be used to validate the requirements.

In addition, for generating software satisfying requirements on fault occurrence or fault-tolerance, **fault-tree** approaches are used to manually construct fault-trees (or equivalent representations) and, through simulation, test coverage of requirements by the generated fault-trees [11].

8.2.1 *Model-Based Methods*

Model-based methods typically use a collection of languages to model various aspects of a CPS. Examples of integrated tools for MBSE are general tools like [12, 13], or [14], which focuses on model based functional safety analysis. One integrated system, COMPASS, has been developed for critical systems such as aerospace and automotive systems [12]. COMPASS input models use the language SLIM, which is a version of the AADL language that has been extended to incorporate behavior and error specifications. The semantics and syntax of SLIM are summarized in [12]. Models are described in terms of a component hierarchy. Components interact with each other by means of ports, which send either discrete events or data values. COMPASS provides a declarative language for fault specification and then tools for safety verification.

8.2.2 *Learning-Based Methods*

Researchers have applied learning to several of the steps of MBSE, with the learning target being based on the MBSE sub-sequence addressed. Two of the most popular targets are the behaviour models and the fault-trees. In general, these purely data-driven approaches still need refinement to improve their performance with respect to manual techniques, and to improve the robustness of the learning to noise and missing data. To date, limited tools are available for design automation in AI-based systems.

Hartsell *et al.* proposes a tool-suite for all aspects of developing CPSs that have Learning-Enabled Components (LECs) [3]. These aspects include architectural modeling, engineering and integration of LECs, including support for training data collection, LEC training, LEC evaluation and verification, and system software deployment. The tool suite focuses on safety modeling and analysis.

Meijer *et al.* focuses on learning finite-state automata to represent behaviour models, using requirements specified in Linear-time Temporal Logic (LTL) as inputs [15]. The LTL formulae are checked on intermediate automata, and potential counterexamples are validated on the actual system. Spurious counterexamples are used by the learner to refine these automata.

A recent trend has focused on using machine learning, and in particular deep learning, to generate CPS systems directly from data [5, 16]. This strand of research focuses on black-box (neural network) representations for models, in contrast to white-box models like automata, e.g., [15].

8.2.3 *Fault Trees*

Fault trees are a well-established and well-understood technique used for evaluating safety/dependability of a wide range of systems: see [11] for a survey. Fault trees typically require significant manual effort both for their generation and analysis, even where software tool support exists. Recent work has focused on automating fault tree synthesis from system models. For example, fault trees can be automatically generated from a range of model languages, such as MATLAB [17] or AADL [18].

[19] has developed an approach for learning fault trees from data. This approach produces results whose performance depends on the percentage of noise in the data: for example, the fault tress perform with around 65% accuracy given up to 3% noise in the data, at a significance level of 0.01. However, the learning algorithm has exponential time complexity since it does an exhaustive search, and it also cannot deal with hidden variables.

8.3 Model-Based Software Engineering

MBSE is a software development process that uses abstraction and automation to improve software-development speed and accuracy by tackling software development complexity [2]. Abstraction is achieved by employing suitable models of (parts of) a software system. Automation systematically transforms these models into executable source code [20].

8.3.1 *MBSE Languages*

We define a model that is used in MBSE as follows:

Definition 8.1. Model [21]. A model is an abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose.

Specifying models can either be done using a general-purpose modeling language (GPML) or a domain-specific language (DSL).

Definition 8.2. Modeling Language [21]. A modeling language defines a set of models that can be used for modeling purposes. Its definition consists of (a) the syntax, (b) the semantics, and (c) its pragmatics.

While modeling languages are usually not tailored to a particular domain but rather address general-purpose concepts (e.g., the UML [22]), a DSL uses model-specific representations:

Definition 8.3. Domain-Specific Language [23]. A DSL is a language that is specifically dedicated to a domain of interest, using representations that enable communication between stakeholders.

A DSL aims to bridge the gap between problem and solution space [23] and consequently is more restrictive than a general-purpose modeling language. DSLs usually drop Turing-completeness, and often allow fully automated formal verification of the (domain-specific) properties of interest. This is hardly feasible using Turing-complete general-purpose programming languages. If a system is grounded on a well-defined theory such as physics, chemistry, or biology, then researchers develop models with syntax similar to the underlying mathematical theory, e.g., using differential equations in the language Modelica (www.modelica.org).

8.3.2 *Traditional MBSE Process*

MBSE typically consists of the following tasks:

(1) **Requirements generation**: Generate a formal representation of the task requirements.
(2) **Model construction**: A DSL is used to represent the entity (e.g., CPS) and its operation within a specific environment. The system structure (the system's components and their interconnections) is often represented using an architecture description language [24], which can be used in conjunction with the DSL to auto-generate a model that can be used for simulation and analysis.
(3) **Verification & validation**: Various properties, e.g., safety, can be verified on the model or generated code.
(4) **Code generation**: Once the CPS has been modeled and verified, automated methods for generating embedded code can be applied to the model.

The traditional approach to MBSE is shown in Fig. 8.1. Here, a manual process is used to construct a model, which is iteratively improved until the model Φ and requirements R are consistent, i.e., $\Phi \wedge R \not\models \bot$, after which it is fielded in the target application. We note that the model is fixed once it is fielded.

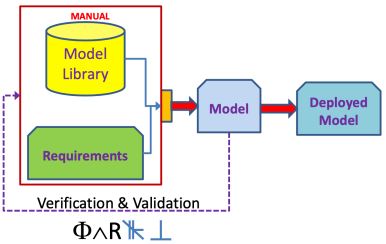


Fig. 8.1 Traditional approach to MBSE.

Definition: **MBSE Task** $[R, M, \mathcal{D}]$. Given a set R of requirements (for functionality/safety/privacy/security etc.), and component model library M, design a system model that can be guaranteed to meet the requirements R using model library M such that $\Phi \wedge R \not\models \bot$.

MBSE models a software system or parts of the system by abstract models, which are then used constructively for different aspects, e.g., functional demonstrations, safety analysis, etc. MBSE tools develop and employ code generators as well as model interpreters to reflect the models' meanings in a system. A code generator takes models as input and produces (parts

of) a software system [25]. Assuming correctness of the code generator, MBSE reduces manual development costs. However, since models typically omit certain details due to model abstractions, the generated code typically has to be manually complemented (e.g., with handwritten code). This can either be done on the generated source code level (e.g., [9]) or on the input model level (e.g., UML/P). Successfully applying constructive MBSE methods requires expertise in (a) application domain and the underlying modeling DSL; and (b) code generator or interpreter development.

8.3.3 *CPS Model Representation*

In this article, we frame our CPS as a dynamical system:

Definition: **CPS model**. We model the physical aspect of a CPS as a dynamical system as follows:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t), u(t), \theta),$$
$$\boldsymbol{y} = h(\boldsymbol{x}(t), u(t), \theta),$$

where

- $\boldsymbol{x} \in \mathbb{R}^n$ is the state vector,
- $u \in \mathbb{R}^m$ are known process inputs (manipulated variables or measured disturbances taking arbitrary values independently of the rest of the variables),
- $\boldsymbol{y}$ is the system observation,
- $\theta \in \mathbb{R}^\rho$ are model parameters (assumed to be constant) and
- $f(\cdot) \in \mathbb{R}^n,\ h(\cdot) \in \mathbb{R}^l$ are nonlinear functions of their arguments.

A CPS also has a cyber element, which in this case is the set of output measurements y and the control settings Ω for the input flows and the valves.

We frame the measured data for our CPS as follows:

Definition: **Measured Data**. We assume that we have measured data $\mathcal{D}$ represented as a collection of pairs (ξ, c), where $\boldsymbol{\xi} \in \Xi$ consist of measured values from a set Ξ, and $\boldsymbol{c} \in \mathcal{C}$ consist of class labels for each measurement.

8.3.4 *Model Development and Validation*

The standard MBSE approach is to develop a model Φ and use verification techniques to show that the model satisfies the requirements R. In this approach, domain experts manually generate a model with parameter set

θ, and then estimate the parameters given data $\mathcal{D}$. The model is viewed as being "complete", in that it captures all necessary aspects of the application system.

In the model-based approach, the model Φ aims to predict the behaviour of the underlying system, which then is used for a top-level task, e.g., alarm generation. Hence the model aims to mimic the system's behaviour, i.e., to develop an accurate CPS model, as in Definition 8.3.3.

In traditional MBSE situations in which the model is validated using theorem proving, e.g., [26], one attempts to show that the model and requirements are consistent. Such model validation is typically performed without the use of data from real-world applications. In this case, if we have a model $\Phi(\theta)$ that is based on a set θ of parameters, we typically fix the parameters for model validation, and use these parameters during operation. The drawback of this approach is when the parameters do not match the real-world system, the software will never perform optimally. Further, if the operational system changes (due to natural degradation or changes in environmental conditions) the model also cannot adjust its performance by updating θ.

8.4 Running Example

This section introduces the phases of MBSE, and illustrates them with a well-known process-control example.

8.4.1 *Process-Control Example: Three Tank System*

To illustrate the concepts of this article, we use a running example of a well-known process-control example, a three-tank system. We show the 3-tank example in Fig. 8.2. The three-tank system is a prototype of many industrial applications in the process industry, such as chemical and petrochemical plants, oil and gas systems. The control of liquid level is a crucial problem in such process industries, and this is the control task on which we focus. While most articles on process control focus on the control issues, here we focus on the model-based methodology for generating the embeddable control and diagnosis software. We use the models developed for simulation and control in the software development process.

The system consists of three identical tanks, T_i, $i = 1, 2, 3$. (We use subscript i to denote tank i.) Pumps are used to provide input flows to tanks T_1 and T_2, with the flows denoted Q_1 and Q_2, respectively. Each

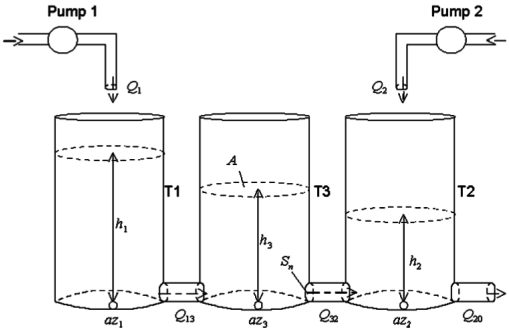


Fig. 8.2 3-tank system with inflows at the two outermost tanks.

input flow (Q_1 or Q_2) can be controlled to a level varying continuously between 0 and a maximum flow Q_{max}. We control flows between tanks using a switching valve for tank i, denoted V_i, $i = 1, 2, 3.$; each valve can be controlled with maximum values of open or closed. The liquid levels h_1, h_2, and h_3 in each tank can be measured with continuous-valued level sensors. The objective is to maintain set-point heights (h_1^*, h_2^*, h_3^*) in each of the tanks, by controlling the inflow rates and valve settings.

In the following, we will briefly illustrate the four steps of MBSE, but focus on model construction, which is the core topic of this article.

8.4.2 *Requirements*

Requirements generation is one of the most important MBSE steps, and generates a formal representation of the task requirements. Requirements generation for process control has been well studied, with a seminal paper being [27]. Recently, researchers have been developing methods to use ML for aspects of requirements generation, e.g., [8]. Reviews of this recent work on software requirements engineering using machine learning techniques are contained in [28, 29], noting that significant work is needed to create industrial-strength engineering tools.

We address the task of developing software for monitoring a hydraulic system for chemical process control. In this process-control system we frame the primary requirement as:

R1 pump two different chemicals into 3 tanks and ensure that the levels in the tanks are maintained at set-points for fixed time periods to ensure proper mixing of the chemicals (nominal operation)

R2 monitor the tank levels

R3 compute a residual in each of 3 tanks, computed as the difference between the measured and predicted levels, with a modelling accuracy level of q%

R4 if the residual exceeds a threshold, isolate the fault (in real time) causing that anomaly

R5 adjust the control system to tolerate any isolated fault, if possible (fault-tolerant operation)

This system is designed to have two main control regimes: (a) nominal operation and (b) fault-tolerant operation. We explicitly use a system model for developing these controllers.

Requirements engineering is a challenging task, and the mapping of requirements into system models is non-trivial. Also, requirements may change over time, for which machine learning is becoming increasingly useful [8]. We assume a complete set of requirements, and leave the application of AI/ML to both requirements and modelling of evolving systems to future work.

We can formally state requirement R_1 as minimizing the difference between the actual ($\boldsymbol{h}$) and set-point ($\boldsymbol{y}$) tank-height values, i.e.,

$$\min_{y(t), t \in [0,...,\tau]} \parallel \boldsymbol{h}(t) - \boldsymbol{y}(t) \parallel, \tag{8.1}$$

where $[0, ..., \tau]$ is the temporal window of interest.

To build a system to satisfy this requirement, we can define a system model Φ and show that the model satisfies requirement $R_1 - R_5$, i.e., the behaviour of the model does not lead to states that violate the level-control, safety and other constraints.

8.4.3 *Model Construction*

To build a system to satisfy these requirements, we can define a system model Φ and show that the model satisfies requirements $R_1 - R_5$, i.e., the behaviour of the model does not lead to states that violate the level-control, safety and other constraints. The most challenging model requirement is to balance the necessary accuracy level with real-time performance, e.g., fault isolation. Even for relatively simple non-linear systems like this tank example, using a first-principles ODE model may not lead to real-time fault isolation. Hence, in this article we focus on the model construction challenges of balancing accuracy with inference efficiency.

8.4.3.1 *Approach*

The classical approach to modelling a non-linear system is using ordinary differential equations (ODEs).

Generating models from data avoids manual model construction, but it still poses research challenges. Even state-of-the-art methods for inducing physics-based models from data still are limited to toy models, e.g., [33].

8.4.3.2 *Modeling Language: ODEs*

We model this non-linear system using ordinary differential equations (ODEs). Due to the difficulty of performing inference on this non-linear system, significant effort has been devoted to creating simplifications of non-linear systems, e.g., [30]. These simplifications include Mixed-Integer Linear Programs [31], smooth Metric Temporal Logics [32], linear approximations [30], among others. We use a linear approximation in this article. This section first describes our non-linear equations, and then a set of linear approximation of these equations.

Nominal System Description Φ_{nom}. For this 3-tank system (with inputs Q_1 and Q_2 at the outermost two tanks, and valves controlling the flows between adjacent tanks), the equations are given by

$$\begin{aligned}
\dot{h}_1 &= \frac{1}{A}[Q_1 - Q_{13} - Q_{1l}] \\
\dot{h}_2 &= \frac{1}{A}[Q_2 - Q_{32} - Q_{20} - Q_{2l}] \\
\dot{h}_3 &= \frac{1}{A}[Q_{13} - Q_{32} - Q_{3l}]
\end{aligned} \tag{8.2}$$

where

$$Q_{ij} = \eta_i V_i sign(h_i - h_j)\sqrt{2g|h_i - h_j|} \tag{8.3}$$

(through Bernoulli's law), η_i is a coefficient summarizing outflow parameters, A is tank area, and V_i is the [0,1] valve setting where 0 denotes closed and 1 open.

Fault System Description Φ_F. To create fault model Φ_F, we extend the equations in Φ_{nom} with fault equations covering possible faults in each state in Q. We assume that the impacts of the pump and valve faults are additive; as a consequence we model each fault independently in the fault model. Under this assumption, we can simulate the impact of multiple-fault scenarios by activating multiple faults at a time.

We assume that we can have faults in the pumps and valves. A pump fails off, such that the flow for pump i, $Q_i = 0$, for $i = 1, 2$. We model

a valve fault as an additive fault with (bounded) parameter $\zeta \in [-1, 1]$. When the valve setting is $V_i \in [0, 1]$, $i = 1, 2, 3$, the faulty setting is given by $(V_i + \zeta) \in [0, 1]$, $i = 1, 2, 3$. For example, if the tank is commanded to be open ($V = 1$) but is stuck shut then we model this using $\zeta = -1$ such that $V + \zeta = 0$.

8.5 AI-Based Framework for MBSE Task

We propose an AI-based framework as an alternative the the traditional MBSE approach, as shown in Fig. 8.3. Here, the V&V process involves the the model Φ, requirements R and *data* $\mathcal{D}$. We can no longer perform logical validation (since we have data), but instead perform an optimisation process $Opt(\Phi, R, \mathcal{D})$ so that we optimise an objective function over $(\Phi, R, \mathcal{D})$. The generated model Φ is updated continuously as the data generated from deployment changes over time.

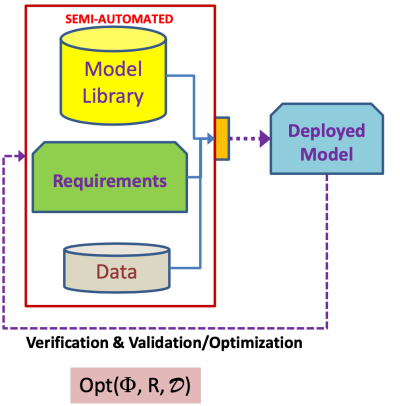


Fig. 8.3 Proposed approach to MBSE.

8.5.1 *Data-Driven MBSE*

Using AI and learning methods can take advantage of the recent availability of vast amounts of data, in order to reduce the onerous manual steps of MBSE. As a consequence, we re-define the MBSE process to incorporate data as follows:

Definition: [Data-driven MBSE Task $\langle R, M, \mathcal{D} \rangle$] Given a set R of requirements (for functionality/safety/privacy/security etc.), a component

model library M, and data $\mathcal{D}$, design a system model that can be guaranteed to meet the requirements R using model library M given data $\mathcal{D}$.

Given a system model Φ with faults, we can then use a range of techniques to verify if Φ satisfies R. In this article we focus on using AI-based methods to automate the model development process.

The MBSE approach of Definition 8.5.1 assumes that requirements are fully verifiable. In reality, requirements may be partially satisfiable. To address such situations, we generalize this definition to incorporate "soft verification" of requirements. To do this, we frame the software generation task as an *optimization problem.* In particular, we focus on a CPS application, where the underlying system has both cyber- and physical-aspects.

8.5.2 *Optimization-Based MBSE*

This section describes an MBSE framework that defines the overall MBSE task in terms of optimizing the outcomes of creating the embedded software, as represented in terms of a system model Φ.

Our aim is to define a CPS that optimizes an objective $\mathcal{J}$ subject to obeying:

- a set of physical constraints, which can be represented by a system model Φ;
- a set $\mathcal{R}$ of requirements;
- data $\mathcal{D}$ that can be used for model development (via learning) and/or model verification.

In this article we focus on the model-construction aspects of MBSE development. In contrast to traditional approaches that create a single model Φ that satisfies the requirements $\mathcal{R}$, we have a more general framework that aims to trade off three MBSE aspects:

- model accuracy–measured using a loss function $\mathcal{L}(\cdot)$;
- model (inference) complexity $\chi(\Phi)$;
- model development cost $\mathcal{C}(\cdot)$.

We presume that we have a model-development function $\varphi(M, \mathcal{D})$ that creates a model given a model library M and data $\mathcal{D}$. We denote the space of possible models as the powerset of φ, namely $\mathfrak{P}^{\varphi}$.

We select a model using a regularization framework, using the equation:

$$\Phi^* = \arg \min_{\Phi \in \mathfrak{P}^{\varphi}} \left[\mathcal{L}(\Phi, \mathcal{D}) \; + \; \lambda_1 \chi(\Phi) \; + \; \lambda_2 \mathcal{C}(\Phi) \right], \tag{8.4}$$

where λ_1 and λ_2 are the regularization weights for the complexity and development costs, respectively.

We now describe these aspects in more detail.

8.5.2.1 *Model Accuracy*

The loss function measures the degree to which a model Φ satisfies the requirements R, denoted as $\mathcal{L}(\Phi, R)$. This measure is a strict generalization of the traditional proof-theoretic notion of requirement validation. We can use a variety of measures, such as squared-error loss between model predictions and actual data.

8.5.2.2 *Model Complexity*

Model complexity measures the "size" of the model, e.g., [34]. We can use three complexity measures: (1) An explicit representation in terms of "degrees of freedom" of a model, e.g. effective number of parameters; (2) code length, a.k.a. "Kolmogorov complexity" (the longer the shortest model code, the higher its complexity, e.g., in bits); (3) information entropy of parametric or predictive uncertainty [35]. The literature contains several measures for each type, e.g., for degrees of freedom we can use AIC or BIC [36].

8.5.2.3 *Model Development Cost*

Another important aspect is the cost of developing a model. With manual construction from a model library, the cost must be estimated from the cost of the model components and the manual generation process. With automated construction, the cost is based on that of the data acquisition and learning process.

8.5.3 *System Verification*

Given a model whose parameters are optimized with respect to data, we then verify that the model does not violate our requirements. Verification can be decomposed into two steps: validating the model with respect to collected data, and then validating the run-time system with regards to the requirements and run-time data. We examine each in turn.

We address situations in which we use data from real-world applications for model generation and validation. To generate a model that is consistent

with regard to data (i.e., make the model best fit the data), we use $\mathcal{D}$ to estimate the parameters θ of Φ.

Given a model whose parameters are optimized with respect to data, we then verify that the model does not violate our requirements. Verification can take many forms; for example, for dynamical systems we may perform reachability analysis to verify the system trajectories do not end up in forbidden states [37]. This consists of using the model Φ together with initial conditions $\boldsymbol{x}_0$ to generate trajectories, which are checked for intersection with forbidden states.

8.6 AI-based MBSE Model Construction Methods

This section reviews approaches (that can be applied to MBSE) for generating models using both (a) physics and/or model-libraries and (b) data. These approaches span a range from purely data-driven approaches (generating black-box models) [38], hybrid approaches combining physics and data (generating grey-box models) [39], to model-driven approaches (white-box) [40]. In the following, we review the data-driven and hybrid approaches, since these are not well-known as MBSE approaches. Based on these modeling approaches, Sec. 8.9 presents an empirical comparison of these model classes.

8.6.1 *Data-Driven Approach*

This approach assumes that we have no manually-generated model, but we generate a model directly from data $\mathcal{D}$. In contrast to the model-based approach, which uses a model Φ aims for tasks like safety analysis or alarm generation, a learning-based approach trains a learning model to directly predict the necessary class labels for the top-level task; in other words, we need not represent the CPS model at all [38]. We generate a classifier Γ with parameters $\boldsymbol{w}$. To learn Γ, we frame our optimization task in quite a different way to the MBSE task: We want to maximize the classification accuracy over the data, in which case we define a loss function as the classification loss: i.e., if our classifier Γ outputs class $\hat{c}$ for data-pair (ξ, c), we minimize the loss as follows:

$$\boldsymbol{w}^* = \arg \min_{w \in \mathcal{P}^w} \sum_{D \in \mathcal{D}} \parallel \hat{\boldsymbol{c}} - \boldsymbol{c} \parallel \tag{8.5}$$

There are many different targets that can be learned, which include:

- **CPS models**: Significant work has been performed to learn CPS models from data, e.g., [41].
- **Fault Trees**: Examples of learning fault trees include [19, 42].

Many papers have applied learning to fault diagnosis of the multi-tank system, e.g., [43]. In order to learn a classifier, we need to generate data that covers nominal as well as all fault conditions.

One of the issues that must be addressed in learning for diagnostics is the *class imbalance problem*, where the data set has an unequal distribution between it's majority (nominal state) and minority (fault state) classes. This imbalance arises since most data that is measured corresponds to nominal cases, i.e., few fault cases are recorded given that faults are rare. Addressing class imbalance problems is important, as it leads to several problems [44], which include:

- learning a classifier to identify the minority class is very complex and challenging since a high-class imbalance introduces a bias in favour of the majority class;
- the predictive accuracy of the classifier (i.e., ratio of test samples for which we predicted the correct class) is not longer a good measure of the model performance, since predicting a nominal state always yields the proportion of nominal in the data set (e.g., 98%), thereby failing in the real task, i.e., correctly predicting fault cases.

In order to address class imbalance in training a classifier, we can use several techniques, such as:

- Using *data augmentation* to artificially increase the proportion of fault cases in the data by perturbing existing fault cases to create novel fault cases;
- Train using a loss function designed for achieving a *good trade-off between sensitivity and specificity*; this will correct "traditional" training methods, that create a classifier that never predicts faults (and hence has a sensitivity of 0 and a specificity of 1).[1]

[1] Sensitivity and specificity, though theoretically independent, typically trade off against each other and are inversely related [45].

8.6.2 *Hybrid Approach: Multi-Fidelity (Surrogate-Based) Optimization*

Multi-fidelity methods leverage models and/or data of multiple levels of fidelity in order to maximize the accuracy of model estimates, while minimizing the cost associated with parameterisation [46]. The model-based methods adopt a set of generative models, while the data-driven methods learn a model, e.g., regression-based model, physical model [47], or a reinforcement-learning model [48].

8.6.2.1 *Model-based Multi-Fidelity Approach*

In situations where a high-fidelity model Φ_H exists but is computationally too expensive to use for inference, surrogate low-fidelity models can be used instead through data-driven tuning [49–51]. A hybrid surrogate model takes advantage of the predictive ability of each individual surrogate model through a weighted-sum combination of the individual surrogate models. In other words, we approximate the system behavior (high-fidelity behavior) using a linear combination of the low-fidelity predictions and a discrepancy function (e.g., polynomial function). The key idea is to consider the low-fidelity model as a basis function in the multi-fidelity model with the scale factor as a regression coefficient. We can perform least-square estimation based on inputs of the low-fidelity model and the discrepancy function. We compute the scale factor and coefficients of the basis functions using linear regression, which guarantees the uniqueness of the fitting process. Besides enabling efficient estimation of the parameters, the proposed least-squares multi-fidelity surrogate can be applied to other tasks.

In this approach, if we have a high-fidelity model Φ_H with output y_H, and a low-fidelity model Φ_L with output y_L, then we can tune the system such that

$$y_H(x) = \boldsymbol{\alpha} y_L(x) \; + \; \delta(x), \tag{8.6}$$

where the regression scale factor $\boldsymbol{\alpha}$ and discrepancy function $\delta(x)$ are obtained through optimization, see [51].

Provan presents an approach that learns a polynomial function for the the scale factor $\boldsymbol{\alpha}$ in conjunction with multiple low-fidelity models $\Phi_{L_1}, \cdots, \Phi_{L_k}$ [52].[2] Feldman *et al.* takes a different approach given as input a collection of component models of differing fidelity: they perform

[2]This approach is similar to ensemble learning, where a weighted combination of multiple classifiers is generated.

an exhaustive combinatorial search over all component combinations to generate the system-level model that best fits the data $\mathcal{D}$ [53].

Various packages that build surrogate models have been developed using different programming languages, such as Scikit-learn in Python [1], SUMO in MATLAB [54], and Surrogate Modeling Toolbox (http://smt.readthedocs.io, [55]).

8.6.2.2 *Data-Driven Surrogate Approach*

Data-driven surrogate models (also known as meta-models, or response surface models) are constructed using *simulation data* from high-fidelity models [56]. Hence, they are approximate *black-box models* that provide black-box relationships between inputs and outputs of a system. We construct data-driven surrogate models by simulating the original (high-fidelity) model at a set of points, called training points, and using the corresponding evaluations to construct an approximate model based on mathematical functions (e.g., Kriging, Gaussian-Process functions, etc.).

The main challenge in data-driven surrogate modeling is how to obtain an approximate model from the simulation data that is as accurate as possible over some domain of interest while minimizing the simulation cost of the data generation. This challenge necessitates the appropriate selection of the structure and complexity of the surrogate model, the number and distribution of the simulation data used for learning the surrogate model, and the validation methods used for estimating the quality of the model. Several tool kits have been developed to enable this approach, e.g., [54].

8.6.3 *Hybrid Approach: Model-Constrained Optimization*

The hybrid approach uses a combination of models and learning for software generation [57]. In this case, we assume that we have an *incomplete model* $\breve{\Phi}$ and data $\mathcal{D}$. We represent the unknown part of the model in terms of a subset $z(t)$ of state variables.

Definition: **Incomplete CPS model** $\breve{\Phi}$. We model an incomplete CPS model as a dynamical system as follows:

$$\dot{x} = f(x(t), u(t), z(t), \theta),$$
$$y = h(x(t), u(t), z(t), \theta),$$

where

- $x \in \mathbb{R}^n$ is the state vector,

- $u \in \mathbb{R}^m$ are known process inputs (manipulated variables or measured disturbances taking arbitrary values independently of the rest of the variables),
- $z \in \mathbb{R}^q$ are algebraic variables (with undefined roles, e.g., representing arbitrary inputs or variables that are functions of other variables),
- $\theta \in \mathbb{R}^\rho$ are model parameters (assumed to be constant) and
- $f(\cdot) \in \mathbb{R}^n,\ h(\cdot) \in \mathbb{R}^l$ are nonlinear functions of their arguments.

Given the partial model, we define the problem as finding the sub-model $z(x, u)$ and parameters θ such that the response of $\breve{\Phi}$ optimally fits the experimental values $\mathcal{D}$. This approach has the potential to provide the best balance between physical models and empirical data, but is computationally intensive. The intractability arises since we are solving a semi-infinite programming problem (an optimization model that has finitely many variables and infinitely many constraints). Recent approaches have explored more tractable optimization algorithms, e.g., [57].

8.7 Running Example: Continued

This section uses our running example to illustrate and compare the different AI-based frameworks.

8.7.1 *Model-based Multi-Fidelity Approach*

To illustrate how surrogate models can be used for the tank system, we can replace the high-fidelity model Φ_H with a low-fidelity (but computationally simpler) model Φ_L as follows. We define a surrogate model Φ_L by replacing the non-linear model (Eqs. (8.2) and (8.3)) with a linear model of the nominal tank system using Eqs. (8.2), except that the outflow from tank i to tank j $(i = 1, 2 \quad j = 2, 3)$ is given by

$$Q_{ij} = \eta_i V_i sign(h_i - h_j) 2g(h_i - h_j). \tag{8.7}$$

For details of such a linearization process, see [58].

We then fit Φ_L to data $\boldsymbol{D}$. In this case, we use simulated data for this fitting. If we simulate the non-linear/linear systems under equivalent initial conditions x_0, we can obtain outputs y_H/y_L, respectively. For such a collection of non-linear/linear simulation outputs $\boldsymbol{D} = \{(\boldsymbol{y}_H, \boldsymbol{y}_L)\}$, we then compute parameters $\boldsymbol{\alpha}, \delta(x)$ to optimize the surrogate model's fit:

$$\boldsymbol{y}_H(x) = \boldsymbol{\alpha}\boldsymbol{y}_L(x) \ + \ \delta(x). \tag{8.8}$$

8.7.2 *Data-Driven Surrogate Approach*

A data-driven surrogate approach, rather than using physics-based surrogate models as just described, learns a low-fidelity model Φ_L from data, and then combines the model's outputs using optimization parameters (e.g., $\boldsymbol{\alpha}$, $\delta(x)$). We can create a neural network surrogate (simulation) model, e.g., as in [59,60], where the model class learned is a Radial Basis Function (RBF) network; alternatively, we can learn a Bayesian network, as in [61].

8.8 MBSE Trade-Off Framework

This section describes our experimental methods, focusing on the models compared and the relative model-construction costs that are key to our comparative analysis.

8.8.1 *Approaches Compared*

In our experiments we compared several approaches, as shown in Table 8.1. We compare the different approaches using the process-control model. We define a gold-standard (high-fidelity) model Φ_H that is manually developed within a traditional model-based framework. We then compare the fault-identification model cost of Φ_H with the that of models that use data for model-generation, using multi-fidelity surrogates, data-driven learning, and model-constrained optimization. We use as our cost metric a measure of manual effort necessary for model-construction.

Table 8.1 Code-generation approaches compared.

Approach	**Notation**	**Diagnostics inference**
Gold standard	Φ_H	Observer-based Engine $\mathcal{E}$
Data-driven (NN)	Φ_{NN}	Neural Network classifier
Model-based Surrogate	Φ_{L_m}	Observer-based Engine $\mathcal{E}$
Data-driven Surrogate	Φ_{L_d}	Observer-based Engine $\mathcal{E}$

We assign cost to the following operations:

- Manual model construction: components and complete models.
- Manual data preparation and class-label assignment.
- Data-driven model construction.

Table 8.2 summarises this comparative cost information. We note that we have assessed these costs based on the models we have generated, and

Table 8.2 Comparative costs for model construction among modeling approaches (all figures in person-hour estimates).

Approach		Library	Model	Data	Total
Manual	HF	100	30	-	130
	LF	60	30	-	90
Machine Learning	HF	-	-	50	50
MB-Surrogate	HF	100	-	-	100
	LF	60	-	-	60
DD-Surrogate	HF	100	-	10	110
	LF	60	-	10	70

we assign all operations (model construction, data preparation, etc.) to have identical cost. It is beyond the scope of this article to assess the costs in a more precise manner.

8.8.2 *Model Library Costs*

We assess costs based on the person-hours required to develop the model libraries. Creating such libraries is labour-intensive, but the benefit is that the models can be re-used. In addition, many commercial libraries exist, both free (e.g., Modelica) and paid, so these costs could be reduced through the use of commercial libraries. We further note that the high-fidelity (HF) models are more expensive to create that the low–fidelity (LF) models.

8.8.3 *Data-Driven Model Costs*

The main costs for data-driven model generation are due to the cost of manual data preparation and class-label assignment (i.e., labeling the test-cases for the data). It is now well known that at least 50% of costs for data mining are accrued for data preparation.

8.8.4 *Comparative Analysis*

Table 8.2 summarizes the comparative costs of the different approaches. The Machine Learning approach is the cheapest, with the other approaches indicating the the low-fidelity model-based methods the next cheapest. Due to re-usability of models, this table may be somewhat misleading, since with new data an entirely new model must be learned using the Machine Learning approach; recent research in transfer learning (e.g., [62]) aims to use prior learned models to avoid this problem.

8.9 Empirical Modeling Cost Comparison

8.9.1 *Empirical Analysis*

We compare traditional and several AI-based (data-driven) MBSE approaches. In the proposed optimisation-focused MBSE process, our optimisation task incorporates both the model construction costs and model accuracy; however, at present these are computed independently. Future work is needed to be able to fold both into a single cost function and subsequently optimize the entire MBSE process.

This section describes our empirical analysis. We compare the different approaches using a process-control model. We define a gold-standard (high-fidelity) model Φ_H that is manually developed within a traditional model-based framework. We then compare the fault-identification performance of Φ_H with the performance of models that use data for model-generation, using multi-fidelity surrogates, data-driven learning, and model-constrained optimization. We develop and train all models using a MATLAB/Simulink platform.

We define an objective function for the experiments as follows:

$$\mathcal{J} = \lambda\Delta + (1 - \lambda)(\tau^* - \tau), \tag{8.9}$$

where

- Δ is the diagnostic accuracy, which we compute in terms of classification error (equation 7 of [63]);[3]
- τ is the diagnostic inference time for the total number of cases;
- τ^* is the target diagnostic inference time;
- λ is a regularization parameter that balances Δ and τ.

Here, we penalize an approach that exceeds the target diagnostic inference time using the $(\tau^* - \tau)$ term. Embedded diagnostics applications typically need timely diagnostics results, leading to application-specific target diagnostic inference times (τ^*). For example, aerospace systems have stringent values of τ^* so that fault-tolerant control (FTC) can be implemented in (near) real-time when a controller uses fault-isolation inputs as part of its control loop, e.g., [65]. In this article, we use a fixed value for τ^*, similar to the maximum fault-isolation requirements for FTC.[4]

[3]Several metrics for diagnostics accuracy have been proposed in the literature, e.g., [63, 64]; any of these metrics can be used for Δ.

[4]Fault isolation for real systems follows a distribution of times, since some faults are computationally harder to isolate than others. It is beyond the scope of this article to

8.9.2 *Data*

We simulate a data set $\mathcal{D}$ from the gold standard model Φ_H by sampling behaviours given nominal and faulty scenarios. We divide $\mathcal{D}$ into training $\mathcal{D}_{train}$ a test $\mathcal{D}_{test}$ subsets, with 90% used for training and 10% for testing.

We denote the system modes $\boldsymbol{\eta}$ as consisting of nominal (no faults, i.e., $\eta = \emptyset$), and faults in (a) pump P_1; (b) pump P_2; (c) valve V_{13}; (d) valve V_{32}; (e) valve V_{20}; and (f) combinations of the fault conditions. We sample modes according the a probability distribution over the fault occurrence, as follows: $P(\eta = \emptyset) = 0.93$; $P(\eta = P_1) = 0.01$; $P(\eta = P_2) = 0.01$; $P(\eta = V_{13}) = 0.01$; $P(\eta = V_{32}) = 0.01$; $P(\eta = V_{20}) = 0.01$.

For each test case, we simulate a behaviour B over $t = 100$ seconds using an initial condition x_0; we inject a fault into some scenarios (according to the mode distribution just described) at $t = 15$ seconds. We measure the tank heights (h_1, h_2, h_3) over $t = 100$ seconds, and our objective is to identify if a fault occurred during the simulation.

8.9.2.1 *Baseline: Model-Based Approach*

This approach uses the model Φ_H, which has had its parameters assigned using the training data $\mathcal{D}_{train}$. Φ_H provides a forecast of expected behaviour of the system, and we use an observer-based diagnosis algorithm [66] to isolate faults.

8.9.2.2 *Data-Driven Approach*

We learn a classifier given the input (B, η). For this application, we have chosen to learn a neural network. We divide the simulation period [0,100] into a sequence of intervals using a temporal window, and we learn the relationships between adjacent intervals to simplify the learning process. Note that this is supervised learning, in that each training case is labeled with the fault class. The output layer consists of the fault class values.

8.9.2.3 *Model-Based Hybrid Approach*

We selected 3 low-fidelity tank models, denoted Φ_{L1}, Φ_{L2}, Φ_{L3}, developed using the approach described in [52]. Then, using the training data $\mathcal{D}_{train}$,

explore such issues, but they could be incorporated into a regularisation task as described in Eq. (8.9).

we learned a polynomial function such that

$$y_H(x) \simeq \sum_{i=1}^{3} \boldsymbol{\alpha}_i y_{Li}(x) \ + \ \delta(x), \tag{8.10}$$

where $y_{Li}(x)$ is the output of Φ_{Li} given input x.

8.9.2.4 *Data-Driven Hybrid Approach*

We used training data $\mathcal{D}_{train}$ to learn a neural network function that generates output $\boldsymbol{y}(\boldsymbol{x})$ at time steps $t = 10, 20, \cdots, 100$.[5] We then used the trained network $\mathcal{D}_{train}$ as the "low-fidelity" model used to generate a polynomial function, such that Eq. (8.10) holds at time steps $t = 10, 20, \cdots, 100$.

8.9.3 *Results and Discussion*

We have averaged our results over a set of 100 test cases, which contained 92 nominal scenarios, 7 single- and 1 multiple-fault scenarios. Table 8.3 summarizes our results.

Table 8.3 Code-generation results, with the best results for each category shown in bold-face.

Approach	**Accuracy** Δ	**time** (s)	$\mathcal{J}: \ \lambda = 0.9$	$\mathcal{J}: \ \lambda = 0.6$
Φ_H	**96**	103.5	**81.1**	26.2
Φ_{NN}	64	**3.8**	62.2	56.9
Φ_{L_m}	85	41.3	77.4	54.5
Φ_{L_d}	78	73.1	21.4	**58.2**

8.9.3.1 *Accuracy*

- Φ_H: The fact that the gold-standard approach did not achieve 100% accuracy is due to the inference engine not being perfect in its diagnostics ability.
- Φ_N: We argue that the poor accuracy of the NN is due to issues with training: there were insufficient training cases ($n = 900$), and significantly more work is necessary to either pre-process the data so that training is simplified, or to tune the architecture. This shows that NN training shifts the burden from manual model generation to (at present) manual pre-processing and architecture tuning.

[5] We trained a deep LSTM network consisting of 6 densely-connected layers. The LSTM was found to capture the temporal aspects of this application and produce better results than atemporal networks.

- **Surrogates**: The model-driven surrogate approach performed the best of the non gold-standard approaches, with the data-driven being slightly worse due to issues with training.

8.9.3.2 *Time*

- Φ_H: The gold-standard approach was computationally the most expensive due to the complexity of ODE inference for diagnostics.
- Φ_{NN}: The NN shows very fast inference, the timings only hampered by the number of parameters for the classifier that need to be processed.
- **Surrogates**: The model-driven surrogate approach performed faster than the gold-standard approach, but still needed to execute costly ODE operations. The data-driven surrogate approach performed faster since it has the fewest parameters in it's pseudo-simulator that must be evaluated.

8.9.3.3 *Combined (Weighted) Objective*

- The optimal value of $\mathcal{J}$ depends on the different parameters used. For $\lambda = 0.9$ (accuracy is weighted highly) the gold-standard approach was optimal. However, if inference speed is also important ($\lambda = 0.6$) the data-driven surrogate approach is optimal.

8.9.4 *Discussion of Trade-Offs*

In this article we are ignoring the cost of parameter estimation or of training the different models.

Modeling Costs: The model-development costs for the NN are significantly higher than those for the other approaches. As a consequence, our results focus on the on-line inference costs and accuracy.

Explainability: The approaches differ considerably in terms of explainability. The gold-standard approach is best in this regard, since all equations are physically well-founded and understandable. The model-driven surrogate approach is also highly transparent, since it weights a collection of physics-based models. The data-driven models, which consist of just weights, do not have explanatory power since they have no physical equations to relate their results to.

8.10 Conclusion

This article has proposed an optimization-based framework for MBSE. Using this approach we have examined traditional (manual) and AI (data-driven) methods for embedded MBSE. We illustrated our framework using an embedded-systems application, in which the aim is to generate embedded code for monitoring a hydraulic tank benchmark such that we isolate faults with high accuracy and inference-time deadlines. The optimization-based framework enables developers to trade off a variety of system parameters in building the embedded code.

This work shows the benefits of AI-based methods for automating MBSE, in that manual model generation can be replaced by data-driven methods. Significant work remains, as this is just a preliminary evaluation of the trade-offs entailed in such automation. Purely data-driven methods are not explainable, and require significant data for training. For diagnostics applications, real-world data for faults in hard to acquire, which may limit the developed classifiers. However, for applications where physics-based models are expensive or impossible to develop, AI-based models show great promise if data is available.

References

[1] O. Lisagor, T. Kelly and R. Niu, Model-based safety assessment: Review of the discipline and its challenges, in *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*. IEEE, pp. 625–632 (2011).

[2] M. Brambilla, J. Cabot and M. Wimmer, Model-driven software engineering in practice, *Synthesis Lectures on Software Engineering* **1**, 1, pp. 1–182 (2012).

[3] C. Hartsell, N. Mahadevan, S. Ramakrishna, A. Dubey, T. Bapty, T. Johnson, X. Koutsoukos, J. Sztipanovits and G. Karsai, Model-based design for cps with learning-enabled components, in *Proceedings of the Workshop on Design Automation for CPS and IoT*. ACM, pp. 1–9 (2019).

[4] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson and I. Crnkovic, A taxonomy of software engineering challenges for machine learning systems: An empirical investigation, in *International Conference on Agile Software Development*. Springer, pp. 227–243 (2019).

[5] J. Zhang and J. Li, Testing and verification of neural-network-based safety-critical control software: A systematic literature review, *arXiv preprint arXiv:1910.06715* (2019).

[6] S. L. Brunton, J. L. Proctor and J. N. Kutz, Discovering governing equations

from data by sparse identification of nonlinear dynamical systems, *Proceedings of the National Academy of Sciences* **113**, 15, pp. 3932–3937 (2016).

[7] F. Bruder and L. Mikelsons, Towards grey box modeling in modelica, in *IFToMM International Symposium on Robotics and Mechatronics*. Springer, pp. 203–215 (2019).

[8] A. Moitra, K. Siu, A. W. Crapo, M. Durling, M. Li, P. Manolios, M. Meiners and C. McMillan, Automating requirements analysis and test case generation, *Requirements Engineering* **24**, 3, pp. 341–364 (2019).

[9] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. M. S. Nazari, K. Müller, A. N. Perez, D. Plotnikov, D. Reiss, A. Roth *et al.*, Integration of handwritten and generated object-oriented code, in *International Conference on Model-Driven Engineering and Software Development*. Springer, pp. 112–132 (2015).

[10] J. Schumann and K. Goseva-Popstojanova, Verification and validation approaches for model-based software engineering, in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, pp. 514–518 (2019).

[11] S. Kabir, An overview of fault tree analysis and its application in model based dependability analysis, *Expert Systems with Applications* **77**, pp. 114–135 (2017).

[12] M. Bozzano, H. Bruintjes, A. Cimatti, J.-P. Katoen, T. Noll and S. Tonetta, Compass 3.0, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, pp. 379–385 (2019).

[13] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi and T. Vardanega, Chess: a model-driven engineering tool environment for aiding the development of complex industrial systems, in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 362–365 (2012).

[14] L. Rogovchenko-Buffoni, A. Tundis, M. Z. Hossain, M. Nyberg and P. Fritzson, An integrated toolchain for model based functional safety analysis, *Journal of Computational Science* **5**, 3, pp. 408–414 (2014).

[15] J. Meijer and J. van de Pol, Sound black-box checking in the learnlib, *Innovations in Systems and Software Engineering* **15**, 3–4, pp. 267–287 (2019).

[16] S. Singaravel, J. Suykens and P. Geyer, Deep-learning neural-network architectures and methods: Using component-based models in building-design energy prediction, *Advanced Engineering Informatics* **38**, pp. 81–90 (2018).

[17] Y. Papadopoulos and M. Maruhn, Model-based synthesis of fault trees from matlab-simulink models, in *2001 International Conference on Dependable Systems and Networks*. IEEE, pp. 77–82 (2001).

[18] A. Joshi, S. Vestal and P. Binns, Automatic generation of static fault trees from aadl models, in *DSN Workshop on Architecting Dependable Systems*. Springer Berlin (2007).

[19] M. Nauta, D. Bucur and M. Stoelinga, Lift: learning fault trees from observational data, in *International Conference on Quantitative Evaluation of Systems*. Springer, pp. 306–322 (2018).

[20] O. Kautz, A. Roth and B. Rumpe, Achievements, failures, and the future of model-based software engineering. (2018).

[21] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. Steel and D. Vojtisek, *Engineering modeling languages: Turning domain knowledge into tools*. Chapman and Hall/CRC (2016).
[22] O. M. Group, *Object Management Group*, http://www.omg.org (2017).
[23] B. H. Cheng, B. Combemale, R. B. France, J.-M. Jézéquel and B. Rumpe, Globalizing domain-specific languages (dagstuhl seminar 14412), in *Dagstuhl Reports*, Vol. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015).
[24] N. Medvidovic and R. N. Taylor, A classification and comparison framework for software architecture description languages, *IEEE Transactions on software engineering* **26**, 1, pp. 70–93 (2000).
[25] A. Roth and B. Rumpe, Towards product lining model-driven development code generators, in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, pp. 539–545 (2015).
[26] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione and R. Spalazzese, Model-driven engineering for mission-critical IoT systems, *IEEE software* **34**, 1, pp. 46–53 (2017).
[27] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth and J. D. Reese, Requirements specification for process-control systems, *IEEE transactions on software engineering* **20**, 9, pp. 684–707 (1994).
[28] M. G. Gramajo, L. Ballejos and M. Ale, Software requirements engineering through machine learning techniques: A literature review, in *2018 IEEE Biennial Congress of Argentina (ARGENCON)*. IEEE, pp. 1–7 (2018).
[29] A. Ahmad, C. Feng, M. Khan, A. Khan, A. Ullah, S. Nazir and A. Tahir, A systematic literature review on using machine learning algorithms for software requirements identification on stack overflow, *Security and Communication Networks* **2020** (2020).
[30] A. C. Antoulas, *Approximation of large-scale dynamical systems*, Vol. 6. Siam (2005).
[31] A. Bemporad and N. Giorgetti, Logic-based solution methods for optimal control of hybrid systems, *IEEE Transactions on Automatic Control* **51**, 6, pp. 963–976 (2006).
[32] Y. V. Pant, H. Abbas and R. Mangharam, Smooth operator: Control using the smooth robustness of temporal logic, in *2017 IEEE Conference on Control Technology and Applications (CCTA)*. IEEE, pp. 1235–1240 (2017).
[33] S.-M. Udrescu and M. Tegmark, Ai feynman: A physics-inspired method for symbolic regression, *Science Advances* **6**, 16, p. eaay2631 (2020).
[34] D. J. Spiegelhalter, N. G. Best, B. P. Carlin and A. Van Der Linde, Bayesian measures of model complexity and fit, *Journal of the royal statistical society: Series b (statistical methodology)* **64**, 4, pp. 583–639 (2002).
[35] M. Höge, On the Way to Appropriate Model Complexity, in *AGU Fall Meeting Abstracts*, Vol. 2016, pp. NG13A–1683 (2016).
[36] K. P. Burnham and D. R. Anderson, Multimodel inference: understanding aic and bic in model selection, *Sociological methods & research* **33**, 2, pp. 261–304 (2004).

[37] V. D'silva, D. Kroening and G. Weissenbacher, A survey of automated techniques for formal software verification, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **27**, 7, pp. 1165–1178 (2008).
[38] F. Vaandrager, Model learning, *Communications of the ACM* **60**, 2, pp. 86–95 (2017).
[39] T. P. Bohlin, *Practical grey-box process identification: theory and applications*. Springer Science & Business Media (2006).
[40] A. M. Madni and M. Sievers, Model-based systems engineering: Motivation, current status, and research opportunities, *Systems Engineering* **21**, 3, pp. 172–190 (2018).
[41] T. Q. Chen, Y. Rubanova, J. Bettencourt and D. K. Duvenaud, Neural ordinary differential equations, in *Advances in neural information processing systems*, pp. 6571–6583 (2018).
[42] A. Ragab, M. El Koujok, H. Ghezzaz, M. Amazouz, M.-S. Ouali and S. Yacout, Deep understanding in industrial processes by complementing human expertise with interpretable patterns of machine learning, *Expert Systems with Applications* **122**, pp. 388–405 (2019).
[43] E. Tafazzoli and M. Saif, Application of combined support vector machines in process fault diagnosis, in *2009 American Control Conference*. IEEE, pp. 3429–3433 (2009).
[44] J. L. Leevy, T. M. Khoshgoftaar, R. A. Bauder and N. Seliya, A survey on addressing high-class imbalance in big data, *Journal of Big Data* **5**, 1, p. 42 (2018).
[45] J. Chubak, G. Pocobelli and N. S. Weiss, Tradeoffs between accuracy measures for electronic health care data algorithms, *Journal of Clinical Epidemiology* **65**, 3, pp. 343–349 (2012).
[46] K. McBride and K. Sundmacher, Overview of surrogate modeling in chemical process engineering, *Chemie Ingenieur Technik* **91**, 3, pp. 228–239 (2019).
[47] A. Cozad, N. V. Sahinidis and D. C. Miller, Learning surrogate models for simulation-based optimization, *AIChE Journal* **60**, 6, pp. 2211–2227 (2014).
[48] M. Cutler, T. J. Walsh and J. P. How, Reinforcement learning with multi-fidelity simulators, in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 3888–3895 (2014).
[49] Y. Yang and P. Perdikaris, Conditional deep surrogate models for stochastic, high-dimensional, and multi-fidelity systems, *Computational Mechanics*, pp. 1–18 (2019).
[50] N. M. Mangan, T. Askham, S. L. Brunton, J. N. Kutz and J. L. Proctor, Model selection for hybrid dynamical systems via sparse regression, *Proceedings of the Royal Society A* **475**, 2223, p. 20180534 (2019).
[51] C. Park, R. T. Haftka and N. H. Kim, Remarks on multi-fidelity surrogates, *Structural and Multidisciplinary Optimization* **55**, 3, pp. 1029–1050 (2017).
[52] G. Provan, Model-based diagnosis using variable-fidelity modeling, in *Workshop on Principles of Diagnosis* (2016).
[53] A. Feldman, G. Provan, R. Abreu and J. de Kleer, Learning diagnosis models using variable-fidelity component model libraries, in *IFAC*, Vol. 48. Elsevier, pp. 428–433 (2015).

[54] D. Gorissen, I. Couckuyt, P. Demeester, T. Dhaene and K. Crombecq, A surrogate modeling and adaptive sampling toolbox for computer based design, *Journal of Machine Learning Research* **11**, Jul, pp. 2051–2055 (2010).

[55] M. A. Bouhlel, J. T. Hwang, N. Bartoli, R. Lafage, J. Morlier and J. R. Martins, A python surrogate modeling framework with derivatives, *Advances in Engineering Software*, p. 102662 (2019).

[56] S. A. Forrester and K. A, *Engineering Design via Surrogate Modelling: a Practical Guide*. Wiley, New York (2008).

[57] J. L. Pitarch, A. Sala and C. de Prada, A systematic grey-box modeling methodology via data reconciliation and sos constrained regression, *Processes* **7**, 3, p. 170 (2019).

[58] M. Iqbal, Q. R. Butt and A. I. Bhatti, Linear model based diagnostic framework of three tank system, in *WSEAS Conference SYSTEMS, WSEAS, CSCC* (????).

[59] A. Kouadri and L. Chiter, A hybrid direct optimization algorithm of neural network based model for a three tank system, *J. Appl. Environ. Biol. Sci.* **6**, 6, pp. 97–107 (2016).

[60] M. Jafari and M. Gomez, Online machine learning based controller for coupled tanks systems, in *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, pp. 163–169 (2019).

[61] W. Zhang, W. Feng, H. Zhao and Q. Zhao, Rapidly learning bayesian networks for complex system diagnosis: A reinforcement learning directed greedy search approach, *IEEE Access* **8**, pp. 2813–2823 (2019).

[62] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang and C. Liu, A survey on deep transfer learning, in *International conference on artificial neural networks*. Springer, pp. 270–279 (2018).

[63] A. Feldman, T. Kurtoglu, S. Narasimhan, S. Poll and D. Garcia, Empirical evaluation of diagnostic algorithm performance using a generic framework, *International Journal of Prognostics and Health Management Volume* **1**, p. 24 (2010).

[64] H. R. Depold, R. Rajamani, W. H. Morrison and K. R. Pattipati, A unified metric for fault detection and isolation in engines, in *Turbo Expo: Power for Land, Sea, and Air*, Vol. 42371, pp. 815–821 (2006).

[65] G. G. Yen and L.-W. Ho, Online multiple-model-based fault diagnosis and accommodation, *IEEE transactions on industrial electronics* **50**, 2, pp. 296–312 (2003).

[66] J. J. Rincon-Pasaye, R. Martinez-Guerra and A. Soria-Lopez, Fault diagnosis in nonlinear systems: An application to a three-tank system, in *2008 American Control Conference*. IEEE, pp. 2136–2141 (2008).

AI for Software Testing

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0009

Chapter 9

On the Application of Machine Learning in Software Testing

Nour Chetouane* and Franz Wotawa†

CD Lab for Quality Assurance Methodologies for Cyber-Physical Systems
Institute for Software Technology, Graz University of Technology, Graz, Austria

**nchetoua@ist.tugraz.at*, †*wotawa@ist.tugraz.at*

9.1 Introduction

Software is running almost everywhere providing the backbone of our society enabling communication between persons almost everywhere on earth, allowing to come up with automated and autonomous driving function aiming at increasing safety and comfort, or providing fast and reliable simulations necessary for drug development. Without software many businesses would not exist anymore in the current form including social networks or search engines allowing a fast access to information for everyone. Considering the still growing importance of software, it is of uttermost importance to assure that the software running is dependable, i.e., trustworthy, safe, secure, reliable, maintainable, and available.

In order to assure that software and systems meet such requirements like dependability, we need to come up with measures allowing to verify these requirements. Because of the growing size of software, its interaction and integration with other software and systems, formally proving meeting given requirements seems to be not a viable solution. Hence, there is a strong need for methods and techniques that allow at least to indicate whether the developed software meets its requirements. Testing, i.e., executing the software considering well-specified prerequisites and checking for conforming to given expectations, provides such means. Although, testing

cannot be used for proving that a given software works correctly, testing provides evidence that this is the case of course depending on the way testing is carried out.

Software testing is an inevitable activity in any current software development process in order to assure quality but requires, in addition, a lot of effort for developing tests and also for carrying them out. In order to reduce testing effort but still providing enough evidence that the test assure meeting given quality criteria, we need to further automate testing. Currently, the degree of automation of test execution in practice is high and still increasing. There are a lot of frameworks helping to carry out tests automatically. However, there are still certain limitations. In case of test generation the situation in practice is different. Most of used test cases have been manually generated and maintained. Therefore, there is a need for providing methods that support test automation.

Machine learning has proved to be successfully applied in many areas like prediction and also classification. Autonomous driving without identifying objects from images often relying on machine learning approaches like deep neural networks, would not have been possible. Hence, there is the question whether machine learning can also be applied in the context of software testing. In this book chapter, we discuss several but not all applications of machine learning to all different aspects of software testing including fault prediction, test case generation, and test suite reduction. In all these application domains, machine learning provides valuable results often outperforming results obtained using other methods and techniques.

We organize this chapter as follows: We first briefly introduce background information regarding testing and machine learning in Sec. 9.2. In Sec. 9.3 we discuss the different research methods and techniques that have been proposed for solving certain testing tasks. The objective behind this section is to provide an overview of the application of machine learning to software testing allowing students, researchers, and practitioners to quickly obtain information of how certain machine learning methods can be applied to solve a particular software testing task. Finally, we conclude this chapter summarizing the findings.

9.2 Background

Before starting summarizing and discussing research methods and techniques for software testing utilizing means of machine learning, we discuss the basics behind software testing and machine learning. For the basic

objectives and ideas behind software testing, we refer the less experienced reader to introductory books on software testing like [1]. For an introduction into machine learning there are many text books available including [2].

9.2.1 *Software testing*

Software Testing is without any doubt one of the most important quality assurance activity to be carried out as part of the software development process. The main objective behind software testing is to reveal faults in programs. This is done via searching for certain input values or sequences of interaction that lead to an unexpected behavior of the program or system. From a general perspective, software testing is the process of evaluating a system (or software) under test (SUT) by observing its execution with the intent of finding deviations between the behavior of the SUT and the expected behavior, which is given accordingly to the SUT's requirements [3].

Edsger W. Dijkstra once mentioned that *testing shows the presence, not the absence of bugs*, which is the main limitation of software testing. Therefore, researchers have been substantially contributing to the research field of software testing to come up with ideas and approaches that allow them to decide when to stop testing having sufficient confidence that the program was thoroughly tested. This has been a motivation for many research studies that resulted in several methods and techniques to serve software testing tasks such as the automation of test oracles, software defect prediction for focusing on the most problematic parts of a program, or test case design.

In the following, we briefly introduce the basic principles behind software testing and categorize this research field, which we use later when discussing the application of machine learning. Software testing basically comprises two parts: (i) test suite generation, and (ii) test suite execution. In the context of testing a test suite that is a set of test cases, where a test case is more or less a specification of inputs or sequences of interaction with the SUT together with a test oracle. The test oracle is a means for deciding whether the SUT's output or behavior deviates from the expected ones, when executing the SUT using the inputs or given interactions. A test oracle can range from a set of expected output values to checking for crashes occurring during execution. A test oracle is basically a mechanism for determining whether a test failed or passed.

Both parts of testing, i.e., generation as well as execution, can be at least partially automated. For test suite generation, there exists various

methods and corresponding tools like model-based testing [4], combinatorial testing [5], or random or fuzz testing [6] to mentioned some of them. For test suite execution there are tools available like JUnit for programs written in Java. However, despite the increasing use of software-test, automation software testing is still considered to be the most challenging and expensive part of software development.

In order to perform sufficient testing during the many changing stages of a software's lifetime, it is required to have an effective test suite as it presents a major factor contributing to the adequacy and facility of the testing process with regard to time and cost as well. Thus, it is significantly important to evaluate test cases quality in terms of their fault detection capability, which gets more challenging as the level of automation increases. In addition, highly automated systems usually require large test suites, which as well tend to grow in size when the software evolves making the execution of the entire test suites very time-consuming and expensive to conduct. In this regard, many research activities in the software testing field have been proposed with the purpose of optimization of test cases generation, test suite reduction, test cases evaluation and prioritization. Several of the proposed approaches, dealing with these topics, have revealed that the use of Machine learning algorithms can be very effective for solving such issues and improving the testing quality in general. For describing the application of machine learning in software testing we make use of the following categories:

- *Software fault prediction* deals with estimating the fault proneness of programs and their parts for gaining information regarding code quality or other aspects that can be used for focusing testing efforts. In this context, someone would spend more time or money in those parts that are more likely to cause a failure.
- *Test oracle automation* captures all activities to automate test oracles or its generation.
- *Test case generation* is the most important challenge of testing comprising methods, techniques and tools for the generation of test suites focusing on failure revealing capacities of tests.
- *Test suite reduction, prioritization and evaluation* are necessary activities to keep the number of tests to be executed low without compromising failure revealing capacities. The objective is to find faults as early as possible not exhausting available budget or time restrictions.

- Under the category *other tasks* we summarize all other activities and tasks to be carried out during testing not captured by the previously introduced categories.

9.2.2 *Machine Learning*

Machine learning (ML) [2] aims at automatically generating models from available data more or less replicating learning processes we see in nature. Like other models the expectation is to use the learned models for understanding principles, doing reasoning, and making predictions using observations. Moreover, when being able to learn from data we may also gain a deeper understanding of the human brain. From a general perspective, ML algorithms are centered around learning a function that maps an input domain (i.e., data points) to an output domain with the goal of achieving a certain task, e.g., making a certain decision in a given situation, or predicting figures like the costs of shares at a stock exchange. The underlying idea behind ML is to allow programs to improve their performance and prediction accuracy mainly through experience, i.e., all the data that becomes available over time. The use of ML is rather appealing because it would allow to obtain models directly from data in an automated way not requiring to handcraft such models.

Over the past decades, ML has proven to be of great practical value for many application. Actually, ML has become a common tool in the majority of tasks requiring information extraction from large data sets. ML algorithms, have found their way, as well, into software development practice as these algorithms offer viable solutions for many software engineering issues. In this chapter of the book, we focus in particular on their major role in serving various purposes in the area of software testing such as detecting potential errors, constructing test oracles, or generating test cases (see [7]).

The two canonical types of ML settings are supervised and unsupervised learning. Supervised learning describes a scenario in which the training data contains significant information. Typically, training examples are provided with labels which are missing in the unseen test data to which the learned model is to be applied. Two classic supervised ML tasks are regression and classification. In unsupervised learning, however, no previous information about the training examples is available, therefore, there is no distinction between training and test data. The learner processes unlabelled input data with the goal of identifying patterns or discovering similarities that can lead to some useful analysis, or come up with a compacted version of

that data. The most prominent example of such a task is data clustering where a set of data is partitioned into subsets of similar objects [8].

In ML, a solution can be essentially created in two ways. Either a learned model is constructed from the data directly, which is used in decision trees induction, or it is obtained using search methods that look for an effective solution among a set of candidate solutions. For example, in the domain of neural networks gradient-descent algorithms are used to search for the neuron weights. Stochastic search algorithms such as genetic algorithms (GA) are often used in machine learning applications. They are considered, i.e., in reinforcement learning algorithms where the learning system performance is evaluated by a fitness metric. Hence, they are effective in situations when the only available information is a measurement of performance (see [9]).

In software testing a lot of different ML algorithms have been already used such as decision trees (DT), artificial neural networks (ANN), Bayesian networks (BayesNet), clustering (e.g., K-means), or support vector machines (SVM). Depending on the task one or another ML method might be the most appropriate one requiring to carry out experimental evaluations. The specific task and as well the setup we are facing also influence the choice of the ML method to be used. In case we are interested in prediction or classification, we may make use of supervised ML methods. If we are interested in investigations regarding certain patterns, we may rely on unsupervised methods like clustering.

9.3 Applications of Machine Learning in software testing

In this section, we discuss selected research papers employing the most common ML algorithms like ANN, DT, SVM, k-means, and others for carrying out software testing tasks such as fault prediction, automating test oracles, designing test cases, evaluating test suites, or reducing and prioritizing test cases. Most of the ML methods are designed to learn a mapping function from input data to outputs in order to make predictions or classifications. Therefore, we describe the majority of selected approaches in terms of inputs that are fed into the ML algorithms, and the outputs that are delivered after the performing learning.

9.3.1 *Machine Learning for software fault prediction*

Software fault prediction is a very useful and important task to estimate the expected delivered quality and maintenance effort before the deployment of software. The main goal of software fault prediction is to track and reduce the number of latent software defects as early as possible in the software life cycle, which allows improving the effective software costs, the reliability of the software to be developed, and the achieved customers' satisfaction (see [7]). In the following, we provide a brief discussion of research work in the category of the application of ML for software fault prediction depicted in Table 9.1.

Table 9.1 Research papers describing the use of ML methods applied for software fault prediction.

Papers	**ML method used**
[10]	k-means clustering
[11]	GA
[12]	Naive Bayesian Network, SVM
[13]	DTR
[14]	ANN
[15]	ANN
[16]	GA
[17]	ANN
[18]	ANN
[19]	Ensemble Algorithm
[20]	Ensemble Algorithm
[21]	ANN, unsupervised Self-Organizing Map (SOM)

[10] showed that unsupervised techniques like clustering can be used for software fault prediction especially when fault labels are not available. In this paper, K-means is used for predicting faults in program modules. Cluster centers were initialized by a quad trees based method. The dimensions and metrics used in the data sets represent several metrics thresholds (i.e. Lines of Code (LoC), Cyclomatic Complexity (CC), Unique Operator (UOp), etc. A threshold vector was used for cluster evaluation. As for each cluster, if any metric value of the centroid data point is greater than the threshold, then, that cluster is labeled as faulty, otherwise it is labeled as non-faulty. After evaluation, the overall error rates of this approach are found to be better in most of the cases, than other existing algorithms using other k-means initialization methods.

GA was applied in [11] to determine most important attributes for predicting faulty modules. Raw data was collected in the form of structured source code of open source systems. The collected data is evaluated

using some of Chidamber and Kemerer (CK) software metrics such as LOC, Weighted Methods per Class (WMC), Response for a class (RFC), etc. After data filtering, the metric values are collected and converted into a binary form (i.e. 0, 1) based on a specific threshold. The transformed values are given afterwards as inputs to the GA. After applying mutation and crossover steps on the given binary inputs, the GA outputs a string of binary values showing the most important attributes corresponding to mostly fault prone metrics (i.e. having the value 1). Then, a detailed analysis of the algorithm performance, is made based on a confusion matrix of the GA prediction outcomes.

GAs were also used in [16] for predicting software errors. In order to forecast an error in an application, the proposed method goes through three main steps: first selection of an appropriate software database to use software indices. Second, application of the GA in order to extract important features and then make use of the GA output to determine the probability of an error in the application. Results showed a good performance of the suggested method in terms of time required for predicting errors, and an error detection rate reaching 95%.

In order to improve the quality and reliability of software, many authors suggested the use of ANNs. In [14], authors developed and trained a software defect prediction model using a feed forward back propagation network, with the goal of reducing the overall cost and development time. The data set used for training the network comprises nine software metrics attributes measured for each phase of twenty genuine software projects such as; requirement stability (RS), design review effectiveness (DRE), process maturity (PM), etc. The ANN takes these software measurements as inputs and predicts four density defect indicators for each phase (i.e. requirement analysis phase, design phase, code phase and testing phase). After the training phase, a predictive analysis of software defects and their threshold is done. The testing results showed the proposed approach to be good and effective as the resulting model was able to detect software defects with very less error rate. In their paper, [15], also train an ANN to identify the error prone parts of the software and predict whether a module is erroneous or not. The ANN has been trained for classifying software modules into groups of faulty and non faulty modules. The dataset used for training and testing the model also represents different software metrics attributes. The experimental results showed that the approach provided a good fit in terms of less error in prediction comparing to existing analytical models.

Similarly, [17] introduced an ANN that solves the software fault

prediction as a binary classification problem that determines the faulty or fault-free status of a software module. Also, [18] presented a non-linear hybrid supervised learning method combining ANN with gradual relational association rule mining for classifying defective and non-defective software entities. As for [21], the authors focused on determining the relationship between object-oriented metrics and fault proneness at the source code class level. A hybrid model was designed, as well, combining an unsupervised SOM algorithm and a multi-layer ANN for prediction of faults occurrence. Six CK metrics have been used as input nodes to the ANN and a prediction rate as the achieved output which refers to a pair of fault and fault free classes. The presented validation results in the above studies proved the associativity of software metrics to the fault proneness.

In [13], the authors investigated the capabilities of DT Regression (DTR) for the prediction of the number of faults in given software modules under two different scenarios; intra-release prediction and inter-releases prediction. For the intra-release prediction scenario, the authors performed a 10-fold cross validation for training and testing the model. For the second scenario, previous software releases were used to build the model. For testing it, authors made use of the current release of the same software. DTR is build using independent variables that are software metrics to predict a numeric outcome which is the number of faults in a software module. The experimental study was carried out relying on fault datasets corresponding to five open-source software projects comprising multiples releases. The results showed that DTR provided significant accuracy for the number of faults prediction across all datasets in both considered scenarios.

[19] applied ensemble classification learners to develop a model for predicting fault proneness. The authors proposed a framework for validating source code metrics using the performance of the classifiers to evaluate each candidate metrics and select the right set of metrics that improve the performance of the fault prediction model. The author combined five classification techniques (i.e. one Logistic regression and four different types of ANNs), then computed their final output. The classification models are trained on a data set including twenty source code metrics and module fault information corresponding to several software projects. The experiment proved that multiple combined learners perform better than independent single model in terms of prediction accuracy and lower costs. Later, in [20], the authors, in addition, proposed linear homogeneous ensemble methods using an extreme learning model consisting of an ensemble of single-hidden layer feed forward NNs which combine all their predictions to get a final fault prediction result.

[12] proposed a feature selection technique that is applied for classification-based fault prediction. The authors suggested to train a probabilistic Naive Bayes model and an SVM classifier on historical data of software. The trained models are used afterwards for predicting faults in software. In this study, the authors make use of two public available data sources; from the NASA IV&V facility and from open source Eclipse Foundation, representing a set of non static code features. Each observation in the data sets represents a software module. It consists of an ID, several static code features such as McCabe, Halstead metrics, and LOC. Plus, an error count which, when equal to 0, indicates that no errors are recorded for this software module, and otherwise it takes 1. The data used for training and validating the models is first selected, then, static code metrics are computed on the software source code, and then saved in an attribute relationship file format (ARFF) which is used as a test set for the models.

9.3.2 *Machine Learning for test oracles automation*

Software testing involves executing a program under test and examining the output of the program whether it conforms with the expectations or not. A test oracle is a mechanism required in functional testing to determine whether a test failed or passed when being executed. The oracle can be a human or a piece of software that often operates separately from the system under test (see [7]). Note that the oracle problem, i.e., finding an appropriate test oracle, is still an important topic of testing. In the following, we discuss papers given in Table 9.2 where ML is used for obtaining a test oracle.

Table 9.2 Papers dealing with the application of ML methods for test oracle automation.

Papers	**ML algorithms used**
[22]	DT(C4.5 algorithm)
[23]	GA, SVM
[24]	ANN
[25]	ANN, DT
[26]	ANN
[27]	ANN
[28]	Knowledge Discovery in Database, AdaBoost, JRip

[23] proposed a novel approach to evolve a test oracle making use of genetic programming and SVMs. The authors aim to model specifically the behavior of programs which process and output sequences of integers.

An input/output list relation language (IOLRL) was designed to formally describe the relations between the input and output lists of these software programs. Plus, a GA was applied to evolve relations in IOLRL that can well distinct passing and failing test cases and encode test cases into bit patterns. These bit patterns and some labelled test cases are employed to train an SVM classifier which is used as a test oracle to verify software behavior.

[24] showed that ANNs can be used for constructing a test oracle to automatically handle the mapping between the input domain and the output domain. After training an ensemble of ANNs until reaching the adequate error rate. The complete test oracle output vector is composed of the results of all NNs. Two industry-sized case studies are used for the evaluation. A mutated faulty version and a fault-free of each case study are created in order to test the capability of the proposed oracle to find injected faults. Afterwards, actual outputs produced by the SUT are decided to be faulty or not compared to the ANNs output using an automated comparator. The results proved this multi-networks oracles detected up to 98% of the injected faults with an accuracy of 98.93%.

A similar approach for black box testing using ANN with a comparison tool was proposed by [26]. The network was trained on randomly generated data and was able to classify the test data with 100% accuracy and basically becomes a simulated model of the software application. When executing the software on real data, the network was able to monitor its behavior. This trained network is assumed to be fault-free. In the evaluation, some faults are inserted to the tested program, and a tool is used to compare the SUT output to the network output and decide if is correct or not. The results of experiments on one of the benchmarks programs showed that 92% of faults were detected by the neural network.

[27] presented a similar methodology for testing real time applications using a Back-propagation ANN. Training input data is generated randomly and then fed to the network. Changes with mutation testing are applied to generate faulty versions of the original program. For each input, a comparison tool is used to evaluate the correctness of the obtained results based on the absolute difference between the ANN output and the corresponding value of the program output. If a large difference is recorded between the two outputs then a program defect is indicated. Back-Propagation learning method ensures that the trained network can be updated by learning new data for evolving versions of the software. However, the author reclaims that this approach could have some limitations in case of larger combinations of inputs and outputs.

[25] provided a method for learning a test oracle utilizing ANN and DT model to generate expected outputs. The DT model can also be used for detecting software faults along with mutation testing. In the evaluation, the authors trained and tested the two models on the well-known Triangle program. The study showed that DTs provide maximum accuracy as a data mining technique for test oracle however the proposed DT based approach can only be used for small programs which take integer input data besides the framework is not reliable enough for non-deterministic software. Similarly, in [22], a test oracle based on DT was proposed for identifying failures using binary classification model in order to improve the testing of software with graphical interfaces such as Mesh simplification. They train a C4.5 DT classifier on available known samples, then let it label unseen test cases produced from other programs.

[28] proposed an approach based on ML for automating test oracle mechanism in software. In this paper, a knowledge discovery process is applied on historical usage data in order to define the topology in relation to the inputs and outputs of the ML algorithm. After the selection of two convenient ML algorithms (i.e. AdaBoost, JRip) and their hyperparameters, each of them is trained to generate an oracle suitable for the SUT. The trained model is then used to classify each of the test cases to be run on the target SUT in one of three classes: "Valid", "Fault" and "Possible Fault". To evaluate this approach, the authors carried out three experiments on a web application; first using randomly inserted failures, second using failures inserted by an expert and thirdly using mutation testing. The results showed an accuracy of 94%, 72% and 98% respectively. However, some limitations related to the generalization of the proposed approach have occurred since the example application used in this experiment has a limited set of functionalities.

9.3.3 *Machine learning for test cases generation*

Test case generation is the process of designing test suites, which is one of the most important parts of software testing aiming at providing test suites having a high fault detection rate. A test suite comprises test cases describing inputs and interactions with the system together with the expected output or behavior of the system. A test case can either pass or fail. It fails if the output or behavior deviates from the expected output or behavior when executing the program using the given inputs or interactions. Otherwise, it passes. In Table 9.3 we list some ML applications used

Table 9.3 Papers introducing ML methods applied for test cases generation.

Papers	**ML algorithms used**
[29]	ANN, GA
[30]	GA
[31]	GA
[32]	GA
[33]	GA
[34]	GA
[35]	ANN, GA

for test case generation, we are going to further discuss. It is worth noting that we also added papers that utilize GA without ML in this section for the sake of completeness.

[29] proposed a novel approach for designing adequate test cases on the basis of software specifications. The approach focuses on creating test cases from output domain instead of the input domain. A neural network was trained to be taken as a function substitute for the SUT with the same number of inputs and outputs. The initial inputs of the NN are generated randomly and its outputs are considered as the actual outputs according to the SUT specifications. Based on the function model, the GA searches for the correspondent test inputs for a given output, applying a series of operations (i.e. reproduction, crossover and mutation). The GA stops when its fitness function reaches the maximum value meaning that the corresponding test inputs have been found. Experiments conducted on two different software programs showed that this proposed approach is promising and effective.

[30] made use of GA for fully automating test case design with more focusing on boundary value analysis tests. GA searches for effective test cases in the input domain of the SUT. The initial population of test cases is generated randomly, the Fitness function used here is measured as the difference of each test case from the boundaries of the variable. For selection criteria, the GA uses the roulette wheel method with respect to the probability distribution based on fitness value. The authors compared their approach to Random Testing (RT) and they observed that not only GA outperformances RT but also the overall quality of software is improved in comparison to using RT. The study confirms that GAs are quite useful for increasing the efficiency and effectiveness of software testing, and hence to decrease the overall development cost for software-based systems.

[31] focused on black-box testing methods such as RT and adaptive RT, aiming to optimize a specific string test case generation. A multi-objective optimization algorithm based on GA is used for generating a diverse and effective set of test cases. Several string distance functions were introduced to compute the length distribution of the string test cases, and for controlling the diversity of test cases within a test set as well. An empirical study was performed on several real-world programs. The results have shown that the generated string test cases outperform test cases generated by other methods.

[32] made use of GA for automatic test cases generation and optimization. The proposed algorithm starts by randomly generating initial input tests and executing them. Only test cases whose path coverage is more than 20% are selected as initial population for the GA. New tests are generated after applying GA operations (Crossover and Mutation). For checking the coverage criteria, a probabilistic based fitness function that uses 0.8 probability for crossover and 0.2 for mutation operation. The algorithm stops when path coverage including statement and branch coverage exceed 95%. This method, based on an optimized fitness function, helps achieve final test suites with 100% path coverage.

[33] focused on GA for optimization because it was successfully used by many researchers for providing a more feasible and reliable test suites. This study proposed an optimization approach for test case generation combining GA and mutation testing. For evaluating the performance of the test cases, the authors made use of the mutation score as a selection criteria for the newly produced test cases. The fitness function of the underlying GA relies on the capability of a test case to kill mutants.

[34] focused on structural testing at the unit level. It presents a framework of test case generation using an improved adaptive GA. The main challenge is to search for a set of test cases that lead to the highest path coverage. The proposed method improves search efficiency by maintaining a high population diversity by dynamically adapting crossover rate and mutation rate. Experimental evaluation of the proposed framework were performed making use of six industrial programs. The results approve that the proposed method is more efficient than existing similar methods and also random generation of test cases for assuring path coverage.

[35] proposed a neural network based test case generation approach for data-flow oriented testing. The authors focused on the data flow testing criterion as coverage objective for test case generation. Therefore every variable definition in the program and its use (DU-pairs) need to be

calculated first. A Back Propagation NN is then trained to simulate a fitness function that is mainly based on the control flow graph of the program. The fitness function checks if the test case covers the given DU-pair. Afterwards, a GA is used to generate test cases to cover all DU-pairs. Thus, the fitness value of the test case can be evaluated with the NN instead of running the instrumented program. The results showed that this approach reduces the total time of test case generation especially in case of large and complex program, when compared with traditional GA based methods.

9.3.4 *Machine learning for test suite reduction, prioritization and evaluation*

In this subsection, we discuss the application of ML for test suite reduction, prioritization and evaluation taking care of the papers given in Table 9.4.

Table 9.4 Some research studies of ML methods applied for test cases reduction, prioritization and evaluation.

Papers	ST tasks	ML method used
[36]	Test cases generation, test cases reduction	ANN
[37]	Test cases reduction	Hierarchical clustering
[38]	Test cases reduction	Cluster analysis
[39]	Test cases reduction	semi-supervised clustering with k-means
[40]	Test case prioritization and selection	K-means, Expectation-Maximization (EM), Incremental Conceptual Clustering (Cobweb), DT
[41]	Test cases reduction	K-means clustering
[42]	Test cases reduction	K-means clustering
[43]	Test cases generation, test suite evaluation	DT for model inference, GA
[44]	Test cases reduction	DT for model inference
[45]	Test cases prioritization	ANN
[46]	Test cases reduction	K-means

When testing programs in practice the question whether the used test suite is effective enough arises. This also tells when to stop testing. So, it is significant to evaluate the quality of the underlying test suite in terms of fault detection for a specific program or SUT. Recently test suite evaluation has become a major focus in software quality assurance research.

For assessing the quality of a test suite, the highest number of existing approaches have focused on the source code level of a system assuming that adequacy of test suites is measured by its ability to execute all statements,

branches, or mutants. Yet, in practice, such syntax based criteria of testing are considered, at best, as minimum requirements for a test set and they are rarely sufficient, and, sometimes misleading even when they are satisfied. An alternative approach has been proposed by [43] in order to overcome the shortcomings of syntax based adequacy metrics, the authors recommend the idea of behavioral coverage which basically consists in inferring a suitable model that reflects the system behavior during the execution of its test cases and can be used to provide a much more reliable assessment of test suites. If the inferred model correctly covers the behavior of a system and is accurate enough, then the test suite can be considered to be adequate. The paper allows the use of ML algorithms such as DT for model inference, it also presents a search-based test generation technique using GA which extends standard syntactic and optimizes the generation of rigorous test sets. The authors used a selection of Java units for the empirical evaluation, the results demonstrate that test cases with higher behavioral coverage significantly outperform current baseline test metrics in terms of detected faults.

Test suite size has a direct impact on the cost and effort of software testing. In this regard several different approaches have been proposed for test suite reduction including test suite minimization, test case selection and test cases prioritization. Test suite minimization is a process of detecting and then removing the obsolete redundant test cases from the test suite. Test case selection consists in choosing a representative subset of test cases from the original test suite that will be used to test the newly introduced parts of the software. Test case prioritization focuses on the identification of the ideal ordering of test cases that maximize desirable test requirements, such as early fault detection. These test suite reduction techniques must preserve the highest possible test suite fault detection capability in addition to minimize the size and the time of the testing process.

[36] presented an automated approach for test cases reduction, mainly based on input-output analysis of the tested software. First, a multi-layer ANN is build with similar characteristics as the SUT such as types and number of inputs and outputs. For obtaining the ANN training set, input data is generated randomly and then fed to the original program which generates the corresponding outputs. During the ANN back-propagation training process, a penalty function is used in order to assign lower weights to less important connections. Then, ANN is pruned by removing edges which have lower weights without affecting the predictive accuracy of the network. Thus, only connections corresponding to the most important

attributes are retained in the network. Then feature ranking is performed by sorting inputs in the order of their importance; according to the set of weights from the pruned algorithm. This helps to identify the most significant attributes contributing to the value of output. Test cases with lower ranked attributes are thus removed. After pruning, rule extraction phase is performed in order to express the Input-Output relationship in the form of If-Then rules. Then, clustering is applied to build equivalence classes for continuous attributes. Afterwards, test cases are generated by making combinations of data values of the inputs. Hence, the reduction of size of the input domain conducts to the reduction in test cases. ANN was also applied as black box testing method for test cases prioritization in [45]. Input data is randomly generated by extracting inputs from design specifications and correspondent outputs are obtained by manually executing the input tests. The ANN is trained to assign priorities to the obtained test cases based on a set of priority assignment rules using design specifications and software requirement specifications. The network output is the priority assigned to corresponding test cases. Experiments have been performed on different NN with 2 to 20 layers, the results show that the NN can be used for effectively predicting a test case priority.

Clustering techniques were shown by several research studies to be very useful for test cases reduction purposes. In [37], authors proposed a clustering based approach to help making a selection of diverse test cases and removing redundancy. The authors introduced a framework which provides a group of test cases comparison metrics which quantitatively compare any random pair of test cases. Besides, a hierarchical clustering algorithm is performed using program profiles and static execution in order to group similar test cases. A specific threshold is used to drive redundancy elimination, test selection and effectiveness of new test cases. The experimental results identified 10–20% of redundant test cases.

In [38] the authors provided a solution to deal with the trade-offs between test suite reduction and fault detection capability in regression test selection. Clustering of test cases execution profiles was performed to group program executions that have similar features which helps to better understand program behaviors so that test cases can be properly selected to reduce the test suite effectively. The results showed that this approach can significantly reduce the size of test suite, on the premise of finding most of fault-revealing test cases.

Later, authors, in [42], also performed a cluster analysis on other different types of structural profiles which consider sequential, relations and

structural information between function calls including function execution sequence, function call sequence and function call tree reduction. The reason behind this study is that authors believe that binary or numeric vector-based methods which consider only the number of times that a function or statement is executed do not always generate satisfying results. K-means cluster analysis was also applied to select a representative subset from the original test suite based on the similarity of profiles of exercised tests.

In [39], semi-supervised clustering was used for the first time to improve clustering results and test selection. Semi-supervised clustering technique aims to derive appropriate information by providing some labelled data in form of constraints. In this paper, the authors introduce a semi-supervised K-means where a limited supervision was provided in the form of constraints derived from previous test results and recorded execution profiles of tests. For each test, a simple execution profile and a function call profile are generated indicating if the corresponding function is called or not during a running test. The experiment results illustrate the effectiveness of this semi-supervised cluster test selection methods.

In [40], both supervised and unsupervised ML algorithms have been applied. In this paper, main goal is to link test results derived from the application of different testing techniques to functional aspects of the program. First, test results are grouped into similar functional clusters which serve as functional equivalence classes. Three famous clustering algorithms k-means, Expectation/Maximization EM and Incremental Conceptual Clustering (Cobweb) are used. After clustering, test inputs are linked to software functional aspects and then can be used for training a DT model, based on C4.5 algorithm, to generate classifiers, in the form of rules, which can serve for many purposes like test cases selection or prioritization. In [41], the authors proposed a mining approach, also based on clustering, to provide better set of test cases. First, a control flow graph is inferred from the tested program, then, independent paths are clustered using k-means algorithm. A reduced test suite is build including test cases represented by each cluster. This approach eliminates test cases which cover similar testing paths of the program.

In [46] the authors proposed a clustering based approach for test suite reduction using k-means algorithm. Representative test cases are selected from each cluster to build a reduced test suite. K-means was combined with binary search in order to look for an appropriate K number of centroids. For evaluating the reduced test suite, the authors compute branch, statement, MC/DC (Modified Condition/Decision Coverage) coverage and mutation

score. The proposed approach can be applied on programs with different types of inputs and outputs (numeric, strings and booleans). A first evaluation have shown an average reduction of 95.9% with same branch coverage as originally but with a small decrease in the mutation score for some examples. Nevertheless, 82.2% reduction was provided with a guarantee of same coverage and mutation score as the original test suite.

DTs were employed by [44] to infer a model from the SUT. The proposed model-based test suite reduction approach was conducted without the need to execute the program. The idea behind is to remove test cases that do not change the learned model by checking the similarity between deduced DTs after each test case removal. This approach maintains almost same level of code coverage and mutation score as the initial test suite.

9.3.5 *Other tasks*

In the following two subsections, we briefly summarize the content of the papers given in Table 9.5, which cannot be assigned to the other software testing categories.

Table 9.5 ML methods and techniques applied in other software testing tasks.

Papers	**Other tasks**	**ML algorithms used**
[47]	Software quality evaluation	ANN
[48]	Software performance prediction	ANN
[49]	Software reliability prediction	ANN
[50]	Software reliability prediction	ANN
[51]	Assessing Software reliability	GA, SVM, BayesNet.
[52]	Test time prediction, test cost estimation	Classification tree
[53]	Cost and Execution effort estimation of testing	ANN, SVM
[54]	Estimation of testing effort by predicting test code size	k-nearest neighbors algorithm (k-NN), BayesNet, DT (C4.5), Random Forest, and ANN

9.3.5.1 *Software Quality Prediction*

Software quality prediction models aim to ensure the reliability of the delivered software and help building products of highest quality by predicting quality factors such as whether components are fault-prone or not. Effective verification and accurate prediction of fault prone modules have a major role of eventually increasing productivity and reducing risk.

[47] introduced an approach for software quality evaluation. The approach is mainly based on an improved back-propagation neural network, which is mainly used for building a comprehensive evaluation model for software quality. The inputs of the neural network are evaluation indicators that reflect different levels of software quality. After training the ANN a knowledge base is iteratively formed that can be used for evaluating the software quality comprehensively.

[48] used ANN for predicting the performance of a software by describing the input-output relationship. The authors proposed to make use of ANNs to learn the correlation between the most effective input factors and the software performance.

The two papers [49] and [50] use ANNs for software reliability prediction. The main idea is to use ANN based models to produce accurate prediction results of software reliability based on fault history data without any further assumptions. The proposed neural network predicts software reliability by learning the dynamic temporal patterns of the fault data.

In [51] the authors try to improve software reliability by accurately predicting errors in short time and low cost. For this, the authors suggest to apply a GA to optimize test data and generate effective test cases. In addition, the authors also make use of other machine learning algorithms such as SVM or Bayesian networks to design models for predicting faults in programs at early stages and to repair them. The discussed results show that software reliability fault prediction depends on the number of defects, which are already present in the software before deployment.

9.3.5.2 *Test cost estimation*

Test cost and execution effort estimation is very much important when managing software projects. The aim of test cost estimation is to provide an accurate prediction of the effort needed to develop and test software systems with respect to time and cost required to complete a defined task in the testing cycle.

In [52], the authors made an experiments in order to investigate whether machine learning techniques can be used to determine important software testing attributes and for predicting testing cost and specifically testing time. They create a classification DT in order to identify relevant factors that affect testing time. After analyzing the DT, they noticed that each node represents a group of programs having similar attributes sharing an average testing time. Therefore, they conclude that classification tree can

be effectively used in estimating the testing time of a new program as well.

In [53], the authors introduced the application of machine learning algorithms for estimation the execution effort of functional testing. In the paper, the authors focussed on SVM to solve non-linear regression problems. In addition, the authors also trained a feedforward multi-layer perceptron network using an asymmetric cost function that helped to build a model that favors overestimation over underestimation.

In [54] different machine learnings techniques such as linear regression, k-NN, Naive Bayes, DT (C4.5), Random Forest, and ANNs were empirically investigated for performing an early prediction of the effort required to test object-oriented software from the perspective of test code size, i.e., the required test lines of code (TLOC). For training the prediction models, the authors used functional requirements, describing use cases because they are the main available input at an early stages of the software development lifecycle. The results showed, that this use-case metrics based approach is more accurate in prediction of TLOC compared to the well-known Use Case Points (UCP).

9.4 Conclusions

As pointed out by [7] many learning methods have known practical problems such as overfitting, local minima, or curse of dimensionality caused by either data inadequacy, noisy data, irrelevant attributes in data, or incorrect domain theory. Therefore, the application of ML in practice may not be that easy requiring a deeper understanding of the methods and their limitations. For known tasks and domains in the case of software testing, we discussed several methods and techniques already available. For new challenges in the area of software testing we have to take care of the limitations behind the ML methods and algorithms. One of the few key limitations of ML algorithms is the fact that for their application we have to make sure the availability of relevant data. In addition, the ML algorithms sometimes need to procure a previous domain knowledge in order to be able to deploy and interpret the outcomes of ML correctly. Moreover, when applying ML we have to assure a high quality of data because the quality has a direct impact on the outcome of analytical learning method. It is also worth noting that one of the main disadvantages of relying on standard ML algorithms is that some of them often require a sort of preprocessing such as data cleaning, handling missing or null data values, data normalization, and removing inconsistencies in data.

As noticed, ML algorithms differ in terms of their function as some of them seem to be more suitable for automating certain software testing tasks than others. For example, ANNs have been widely used for solving problems related to test oracle automation, predicting faults, prediction of testing costs and software reliability. Their main advantages are their robustness to errors in training data and their ability of learning complex functions like non-linear and continuous functions. However, they are commonly known as black box systems because interpreting what a neural network has learned is not that straightforward and can hardly be interpreted by users directly. In addition, ANNs are known to have slow training and convergence processes requiring adequate computational resources. Besides this ANNs lead to multiple local minima in error surface and suffer from overfitting (see [7]).

GAs have been used more often for test case generation and optimization. They are recognized to be very much suitable for tasks that require an approximation of complex functions. GAs are most efficient in a search space where little information is provided, and they usually only require an evaluation function rather than the availability of a large set of data. Yet, one of their limitations is that they are computationally expensive and time-consuming especially when the solution space is continuous. In addition, most of approaches using GAs in test case generation, need to run the tested program for each generated test case to evaluate its fitness value which costs a lot and consumes a lot of running time, especially for large scaled programs [35]. We have also noticed the current use for clustering techniques such as k-means in test suite reduction. This might be because clustering is a common solution when no labelled data is available, which is usually quite hard to afford in case of real-world data. [55] stated that it is not always easy to truly take advantage of the benefits of ML algorithms without fully considering their assumptions and implications.

References

[1] G. J. Myers, *The Art of Software Testing*, 2nd edn. John Wiley & Sons, Inc. (2004).
[2] T. Mitchell, *Machine Learning*. McGraw Hill (1997).
[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press (2016).
[4] I. Schieferdecker, Model-based testing, *IEEE Software* **29**, 1, pp. 14–18 (2012).
[5] D. Kuhn, R. Kacker and Y. Lei, *Introduction to Combinatorial Testing*,

Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis (2013).

[6] Y. Koroglu and F. Wotawa, Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing, in *Proceedings of the 14th International Workshop on Automation of Software Test*, AST '19. IEEE Press, pp. 28–34 (2019), doi:10.1109/AST.2019.00010, `https://doi.org/10.1109/AST.2019.00010`.

[7] D. Zhang and J. Tsai, Machine learning and software engineering, *Software Quality Journal - SQJ* **11**, pp. 22–29 (2002), ISBN 0-7695-1849-4, doi:10.1109/TAI.2002.1180784.

[8] R. Bekkerman, M. Bilenko and J. Langford, *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press (2011).

[9] J. Shapiro, *Genetic Algorithms in Machine Learning*. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-44673-6, pp. 146–168 (2001), ISBN 978-3-540-44673-6, doi:10.1007/3-540-44673-7_7, `https://doi.org/10.1007/3-540-44673-7_7`.

[10] P. S. Bishnu and V. Bhattacherjee, Software fault prediction using quad tree-based k-means clustering algorithm, *IEEE Transactions on knowledge and data engineering* **24**, 6, pp. 1146–1150 (2011).

[11] A. Puri and H. Singh, Genetic algorithm based approach for finding faulty modules in open source software systems, *International Journal of Computer Science and Engineering Survey* **5**, 3, p. 29 (2014).

[12] K. Sankar, S. Kannan and P. Jennifer, Prediction of code fault using naive bayes and svm classifiers, *Middle-East Journal Of Scientific Research* **20**, 1, pp. 108–113 (2014).

[13] S. S. Rathore and S. Kumar, A decision tree regression based approach for the number of software faults prediction, *ACM SIGSOFT Software Engineering Notes* **41**, 1, pp. 1–6 (2016).

[14] T. Sethi *et al.*, Improved approach for software defect prediction using artificial neural networks, in *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE, pp. 480–485 (2016).

[15] D. Mundada, A. Murade, O. Vaidya and J. Swathi, Software fault prediction using artificial neural network and resilient back propagation, *International Journal of Computer Science Engineering* **5**, 03 (2016).

[16] F. S. Fazel, A new method to predict the software fault using improved genetic algorithm, *Bull. la Société R. des Sci. Liège* **85**, pp. 187–202 (2016).

[17] M. Owhadi-Kareshk, Y. Sedaghat and M.-R. Akbarzadeh-T, Pre-training of an artificial neural network for software fault prediction, in *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, pp. 223–228 (2017).

[18] D.-L. Miholca, G. Czibula and I. G. Czibula, A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks, *Information Sciences* **441**, pp. 152–170 (2018).

[19] L. Kumar, S. Rath and A. Sureka, An empirical analysis on effective fault

prediction model developed using ensemble methods, in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, pp. 244–249 (2017).

[20] P. R. Bal and S. Kumar, Extreme learning machine based linear homogeneous ensemble for software fault prediction, in *ICSOFT*, pp. 103–112 (2018).

[21] C. Viji, N. Rajkumar and S. Duraisamy, Prediction of software fault-prone classes using an unsupervised hybrid som algorithm, *Cluster computing* **22**, 1, pp. 133–143 (2019).

[22] W. K. Chan, J. C. Ho and T. Tse, Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs, *Software Testing, Verification and Reliability* **20**, 2, pp. 89–120 (2010).

[23] F. Wang, J.-H. Wu, C.-H. Huang and K.-H. Chang, Evolving a test oracle in black-box testing, in *International Conference on Fundamental Approaches to Software Engineering*. Springer, pp. 310–325 (2011).

[24] S. R. Shahamiri, W. M. Wan-Kadir, S. Ibrahim and S. Z. M. Hashim, Artificial neural networks as multi-networks automated test oracle, *Automated Software Engineering* **19**, 3, pp. 303–334 (2012).

[25] A. Singhal, A. Bansal *et al.*, Generation of test oracles using neural network and decision tree model, in *2014 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)*. IEEE, pp. 313–318 (2014).

[26] N. Majma and S. M. Babamir, Software test case generation & test oracle design using neural network, in *2014 22nd Iranian Conference on Electrical Engineering (ICEE)*. IEEE, pp. 1168–1173 (2014).

[27] V. Sathyavathy, Evaluation of software testing techniques using artificial neural network, *Int. J. Electr. Comput. Sci* **6**, 3, pp. 20617–20620 (2017).

[28] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago and M. Souza, A machine learning approach to generate test oracles, in *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, pp. 142–151 (2018).

[29] R. Zhao and S. Lv, Neural-network based test cases generation using genetic algorithm, in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. IEEE, pp. 97–100 (2007).

[30] R. Kumar, S. Singh and G. Gopal, Automatic test case generation using genetic algorithm, *International Journal of Scientific & Engineering Research (IJSER)* **4**, 6, pp. 1135–1141 (2013).

[31] A. Shahbazi and J. Miller, Black-box string test case generation through a multi-objective optimization, *IEEE Transactions on Software Engineering* **42**, 4, pp. 361–378 (2015).

[32] R. Khan, M. Amjad and A. K. Srivastava, Optimization of automatic generated test cases for path testing using genetic algorithm, in *2016 Second International Conference on Computational Intelligence & Communication Technology (CICT)*. IEEE, pp. 32–36 (2016).

[33] A. Mateen, M. Nazir and S. A. Awan, Optimization of test case generation using genetic algorithm (ga), *arXiv preprint arXiv:1612.08813* (2016).

[34] Z. Ma, Y. Zhang, Q. Li, M. Xu, J. Bai and S. Wu, Resveratrol improves alcoholic fatty liver disease by downregulating hif-1α expression and mitochondrial ros production, *PloS one* **12**, 8 (2017).

[35] S. Ji, Q. Chen and P. Zhang, Neural network based test case generation for data-flow oriented testing, in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, pp. 35–36 (2019).

[36] P. Saraph, A. Kandel and M. Last, Test set generation and reduction with artificial neural networks, in *Artificial Intelligence Methods in Software Testing*. World Scientific, pp. 101–132 (2004).

[37] V. Vangala, J. Czerwonka and P. Talluri, Test case comparison and clustering using program profiles and static execution, in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 293–294 (2009).

[38] C. Zhang, Z. Chen, Z. Zhao, S. Yan, J. Zhang and B. Xu, An improved regression test selection technique by clustering execution profiles, in *2010 10th International Conference on Quality Software*. IEEE, pp. 171–179 (2010).

[39] S. Chen, Z. Chen, Z. Zhao, B. Xu and Y. Feng, Using semi-supervised clustering to improve regression test selection techniques, in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, pp. 1–10 (2011).

[40] A. R. Lenz, A. Pozo and S. R. Vergilio, Linking software testing results with a machine learning approach, *Engineering Applications of Artificial Intelligence* **26**, 5–6, pp. 1631–1640 (2013).

[41] B. Subashini and D. JeyaMala, Reduction of test cases using clustering technique, *International Journal of Innovative Research in Science, Engineering and Technology, 2014 International Conference on Innovations* **3**, 3, pp. 1993–1996 (2014).

[42] R. Wang, B. Qu and Y. Lu, Empirical study of the effects of different profiles on regression test case reduction, *IET Software* **9**, 2, pp. 29–38 (2015).

[43] G. Fraser and N. Walkinshaw, Assessing and generating test sets in terms of behavioural adequacy, *Software Testing, Verification and Reliability* **25**, 8, pp. 749–780 (2015).

[44] H. Felbinger, F. Wotawa and M. Nica, Test-suite reduction does not necessarily require executing the program under test, in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, pp. 23–30 (2016).

[45] H. Bhasin, E. Khanna and K. Sharma, Neural network-based automated priority assigner, in *Proceedings of the Second International Conference on Computer and Communication Technologies*. Springer, pp. 183–190 (2016).

[46] N. Chetouane, F. Wotawa, H. Felbinger and M. Nica, On using k-means clustering for test suite reduction, in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 380–385 (2020).

[47] A. Wang, L. Guo, Y. Chen, J. Wang and Y. Song, Comprehensive evaluation of software quality based on lm-bp neural network, in *2017 International*

Conference on Dependable Systems and Their Applications (DSA). IEEE, pp. 162–167 (2017).

[48] A. Sabbeh, Y. Al-Dunainawi, H. Al-Raweshidy and M. F. Abbod, Performance prediction of software defined network using an artificial neural network, in *2016 SAI Computing Conference (SAI)*. IEEE, pp. 80–84 (2016).

[49] P. Roy, G. Mahapatra, P. Rani, S. Pandey and K. Dey, Robust feedforward and recurrent neural network based dynamic weighted combination models for software reliability prediction, *Applied Soft Computing* **22**, pp. 629–637 (2014).

[50] S. Ramasamy and I. Lakshmanan, Application of artificial neural network for software reliability growth modeling with testing effort, *Indian Journal of Science and Technology* **9**, 29 (2016).

[51] R. Jain and A. Sharma, Assessing software reliability using genetic algorithms, *The Journal of Engineering Research [TJER]* **16**, 1, pp. 11–17 (2019).

[52] T. J. Cheatham, J. P. Yoo and N. J. Wahl, Software testing: a machine learning experiment, in *Proceedings of the 1995 ACM 23rd annual conference on Computer science*, pp. 135–141 (1995).

[53] D. G. e Silva, M. Jino and B. T. de Abreu, Machine learning methods and asymmetric cost function to estimate execution effort of software testing, in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, pp. 275–284 (2010).

[54] M. Badri, L. Badri, W. Flageol and F. Toure, Investigating the accuracy of test code size prediction using use case metrics and machine learning algorithms: An empirical study, in *Proceedings of the 2017 International Conference on Machine Learning and Soft Computing*, pp. 25–33 (2017).

[55] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias and M. P. Guimaraes, Machine learning applied to software testing: A systematic mapping study, *IEEE Transactions on Reliability* **68**, 3, pp. 1189–1212 (2019).

https://doi.org/10.1142/9789811239922_0010

Chapter 10

Creating Test Oracles Using Machine Learning Techniques

Rafig Almaghairbe[a] and Marc Roper[b]

[a] *Department of Computer Science, University of Omar Al-Mukhtar, Derna, Libya*
rafig.almaghairbe@omu.edu.ly
[b] *Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK*
marc.roper@strath.ac.uk

10.1 Introduction

Automating aspects of software testing process can be a key factor in reducing the costs of software development projects. Consequently, researchers in software testing have developed ways to automatically generate and execute test cases, as well as maintain and manage test suites. However, the generation of a test oracle (a mechanism to determine whether the output associated with an input is correct or incorrect) is a relatively neglected area of research. Whilst many tools have been developed to generate test inputs [1], few tools exist to build automated test oracles, making the process of checking test outputs primarily human-centred and as a result expensive and possibly error-prone [2].

Proposed approaches to generate test oracles vary from the inexpensive and ineffective (e.g. implicit oracles) to effective but very costly (e.g. specified oracles). Implicit oracles are easy to construct at practically no cost (e.g., the work of Carlos and Michael [3]), and usually perform well with

general errors like system crashes, null pointer dereferences or unhandled exceptions, but are not applicable for semantic and complex failures [4]. On the other hand, specified oracles can be obtained from formal specifications, and are effective in revealing failures, but defining and maintaining such specifications is a challenging task and hence such specifications are very rare [4].

In recent years, researchers have tried to strike the balance between these approaches and develop a technique which combines the effectiveness of specified oracle and the cost of an implicit one by using machine learning and data mining approaches, in particular anomaly detection, to automatically identify failing tests. The purpose of this chapter is to re-evaluate and analyse the work on test oracles built using such machine learning techniques, and also to identify the properties of automated/semi-automated test oracles required for them to be practically usable. Researchers in the software testing community can use these properties as criteria to evaluate test oracles based on machine learning techniques.

The chapter has been motivated by the following research questions:

- Question 1:
  (a) What anomaly detection approaches (machine learning, data mining etc.) have been used to build automated test oracles?
  (b) Which of the variety of anomaly detection approaches are considered to be the most appropriate for automated test oracles?
  (c) What is the effectiveness of anomaly detection strategies (classification, clustering etc.) for the creation of automated test oracles?
- Question 2:
  (a) What data from software systems have been used to build anomaly detection models for automated test oracles?
  (b) What data from software systems are reported to provide the anomaly detection techniques with the best chance of building an effective automated test oracle?
  (c) How has this data been transformed to suitable set of feature vectors?
- Question 3:
  (a) What types of software faults have been used on the empirical studies of automated test oracles by using anomaly detection techniques?
  (b) What classes of faults are reported to be the most frequently detected via anomaly detection techniques?

(c) Is there any relationship between specific classes of faults and the success of anomaly detection approaches?

The remainder of the chapter is organised as follows: In the next section, a background on test oracles is presented. Section 10.3 provides an overview of the related work. Section 10.4 discusses work that addresses the problem of test oracles creation based on machine learning techniques and answers the research questions identified above. Section 10.5 reports the required properties of test oracle techniques. A road map for further research direction is outlined in Sec. 10.6. Section 10.7 summarises and presents the conclusions.

10.2 Background on Test Oracles

An oracle can be defined as a mechanism that determines and judges whether a system's test results have passed or failed [5]. This function can be carried out by the tester (human oracle), or by automated/semi-automated means. Memon *et al.* [6] defined two important parts of a test oracle: oracle information that represents expected output, and an oracle procedure that compares the oracle information with the actual output. Shahamiri *et al.* [7] summarised the test oracle process as follows: (1) generate expected outputs; (2) save the generated outputs; (3) execute the test cases; (4) compare expected and actual outputs; (5) decide if there is a fault or not. It is worth pointing out that the test case execution activity does not form part of the test oracle, but it is part of the oracle process.

Ye *et al.* [8] have characterised a perfect and complete automated test oracle as follows: (1) it should have source of information which makes it possible to produce a reliable and equivalent behaviour to the software under test (SUT); (2) it should accept all entries for the specified system and always produce the correct result; (3) it should have the answers to the data which is actually used in the test. Again a point worth noting is that the anomaly detection approaches discussed in this chapter are unlikely to meet the first of these criteria.

A traditional and generic test oracle structure can be seen in Fig. 10.1 [9]. In this scenario, the test oracle accesses the set of data required to evaluate the correctness of the test output. The set of data can be obtained from the specification of the SUT and holds enough information to support the oracle's final decision. The following subsections introduce several structures of test oracle functions using different sources of information.

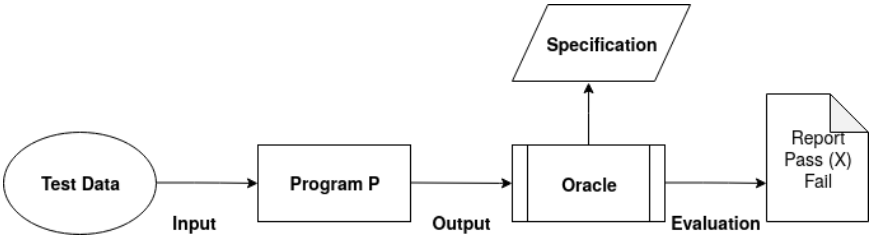


Fig. 10.1 Generic Test Oracle Structure.

10.2.1 *Test Oracles Based on Individual Test Cases*

The development of unit testing frameworks allows the developer to specify the pass/fail conditions for an individual test. This approach is illustrated in Fig. 10.2 which shows how the test case itself carries the expected results in order to decide the correctness of the SUT [10]. The tester can implement this oracle by using one of the various frameworks that are known collectively as an "xUnit" family which support unit testing for a range of programming languages [11] (e.g., "JUnit" is an "xUnit" framework for Java). Testers develop test oracles in their code by inserting assertions in a program to check unit or partial results. This oracle still demands a lot from the developer in that test cases need to be hand coded and acceptable results clearly specified. An example of such oracle is shown in the following piece of code:

```
public void testBOOKInLibrary ( ) {
// A test oracle to check the correctness of the
// method "boolean Library.checkByTitle(String)"
Library library = new Library ( );
boolean search = library.checkByTitle ("Data Mining");
assertEquals (true, search);
}
```

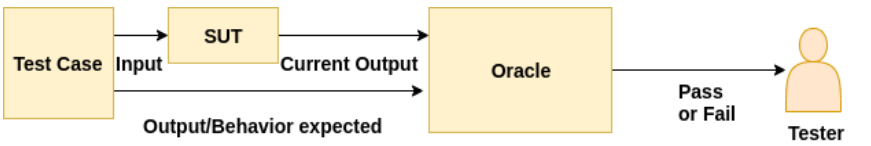


Fig. 10.2 Test Oracle Structure Using Expected Output Behaviours.

Testers can use their own knowledge about the SUT to check if outputs meet the SUT specification. This test oracle is known as a human oracle (Fig. 10.3) [12]. A human oracle suffers from several disadvantages; most notably, it is likely to be error-prone and also slower than an automated check, which may restrict its application to only trivial input/output behaviour.

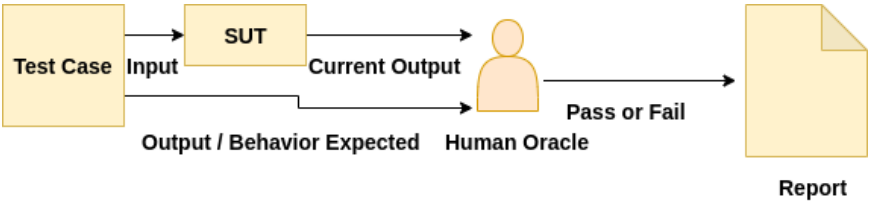


Fig. 10.3 Test Oracle Structure Using Human Oracles.

10.2.2 *Test Oracles Based on Formal Specifications*

Test oracles can be generated from formal models or specifications (Fig. 10.4) [13]. In this scenario a test oracle can be automated when a mathematical model (e.g. Finite State Machine (FSM) or Petri net) of the SUT is available for testers. Test oracles based on formal models or specifications are effective in identifying failures, but defining and maintaining formal specifications is expensive to the point that such specifications are very rare.

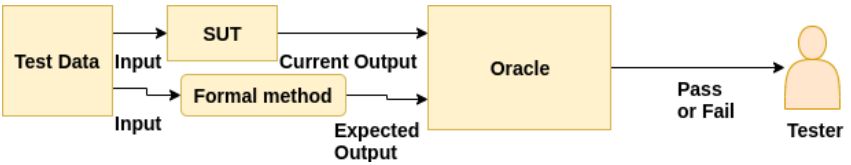


Fig. 10.4 Test Oracle Structure Using Formal Model Specification.

Figure 10.5 illustrates the structure test oracles that derive expected outputs of the SUT from test data inputs [14]. This could be possibly made by using another reference version of the SUT (e.g. an earlier version) to generate outputs from, and then the tester can build a test oracle to compare those outputs and the current outputs. In this case, testers must assume that the version used (the reference program) meets all

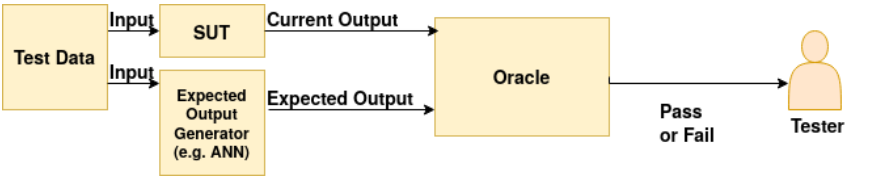


Fig. 10.5 Test Oracle Structure Using Test Data.

specifications of the SUT. This type of test oracle is widely used in the case of regression testing and mutation testing, but is not sufficient in the general case.

The oracle problem usually occurs when it is difficult to interpret test results [15]. In some cases, it is extremely difficult to predict expected behaviours of the SUT to be compared against current behaviours (this depends on the SUT) [16]. Failures can be manifested under different circumstances which makes checking the results complex or impossible to be performed [17]. Some SUTs produce outputs in very complex formats such as images, sounds or virtual environments which make the oracle problem very challenging [16].

10.3 Related Work

The automatic generation of test oracles is an important problem in software testing, but has received considerably less attention compared to other problems such as the generation of test cases. There have been three extensive reviews of topics relating to test oracles. The first by Baresi and Young [2] covered four important topics in the test oracle area: assertions, specification, state-based conformance testing and log file analysis. The second by Pezzé and Zhang [18] discussed the main techniques used to develop automated test oracles based on the available sources of information. In their survey, the source of information for test oracles was classified either as the software specification (e.g. the type of formal model specification: state-based, transition-based, history-based or algebraic) or as the program code (e.g. values from other versions, results of program analysis, machine learning models and metamorphic relations). The third by Barr *et al.* [4] classified the existing literature on test oracles into three broad categories: specified oracles, implicit oracles, and derived oracles. Specified oracles are test oracles obtained from formal specification of the system behaviour. For instance, Doong and Frankl developed the ASTOOT tool

which generates test suites along with test oracles from algebraic specifications [19]. In their work, test oracles generated by the ASTOOT tool may then be used to verify the equivalence between two different execution scenarios. Specified oracles are effective in finding system failures but their success depends heavily on the availability of a formal specification of the system behaviour. However, the vast majority of systems lack an accurate, complete and up-to-date machine readable specification which limits their applicability.

Implicit oracles are generated without requiring any domain knowledge or formal specification and hence can be applied to all runnable programs. For example, in the fuzzing approach proposed by Miller *et al.* [20], the main principle is to generate random inputs and attack the system to find faults which cause the system to crash. If a crash is spotted then the fuzz tester reports the crash with the set of inputs or input sequences that caused it. The fuzzing approach is well used in the security vulnerabilities detection area such as buffer overflows and memory leaks etc. but relies on the consequences of an error being easily detectable (e.g. in the form of a system crash) so has limited general applicability.

Derived oracles are built from properties of the SUT, or several artefacts other than the specification (e.g. documentation and system execution information), or other versions of the SUT. For instance, metamorphic testing has been used to test search engines such as Google and Yahoo [21]. The BERT tool is another example of a derived oracle which can be used to identify behavioural differences between two versions of a program by using dynamic analysis [22].

Each oracle category (specified, implicit and derived) could merit an entire survey in its own right. This chapter is focused on test oracles generated using machine learning techniques (which fall into the category of derived oracles as they are typically created from system executions). Therefore, the chapter is structured according to learning strategies: supervised learning (e.g. techniques that build more on earlier versions of the SUT), semi-supervised learning (e.g. approaches that label a small number of observations and use this to seed the creation of a more complete oracle), and unsupervised learning (e.g. techniques that are based upon clustering similar results and detecting anomalies).

10.4 Test Oracles Based on Machine Learning Techniques

Chandola *et al.* defined anomaly detection as a matter of spotting patterns in data that correspond to abnormal behaviour [23]. This concept is illustrated in Fig. 10.6 where N represents regions of normal behaviour, whereas the points labelled O represent the anomalous data. The work covered in this section aimed at investigating whether software bugs generate a non-conformant pattern of behaviour that can be distinguished from the conformant or normal behaviour — in other words, in Fig. 10.6 do the groups marked N corresponded to passed tests and those marked O with failures? If this is the case then the possibility of detecting bugs automatically can be raised.

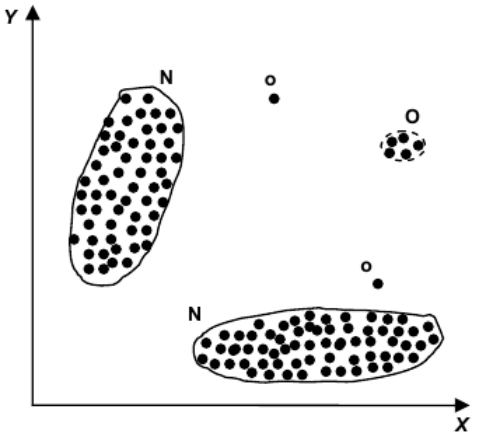


Fig. 10.6 Principle of Anomaly Detection.

The main concept behind generating automated test oracles based on machine learning techniques is to detect unexpected patterns (faulty behaviour) in a large set of observations, events or items [23]. Figure 10.7 shows the principles of using machine learning techniques to automatically cluster or classify (depending on the machine learning strategy employed) passing/failing outputs. The program under test is run on a set of inputs which will generate outputs and optional traces, and may encounter bugs in the program (the *s in the figure). The pass/fail status of the outputs is unknown and the aim is to automatically distinguish between these using machine learning. The application of anomaly detection strategies in

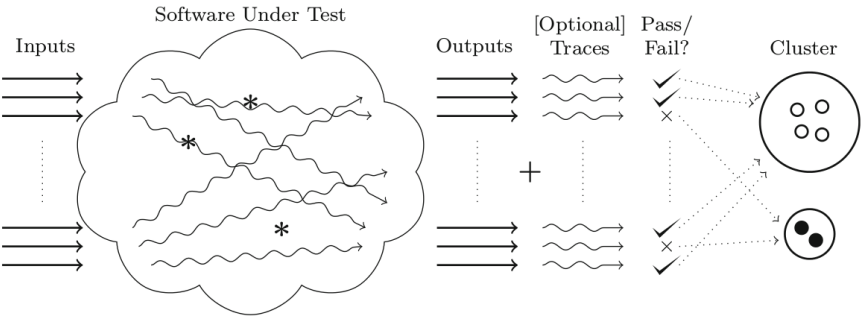


Fig. 10.7 Overview of Test Oracle Based on Machine Learning Strategy.

this context has not been extensively investigated. The current work can be divided into three main categories of learning techniques: supervised, semi-supervised and unsupervised.

10.4.1 *Test Oracles Based on Supervised Learning Techniques*

Supervised learning techniques assume the availability of a training data set which has labelled instances for normal as well as anomaly classes and is therefore the least generally applicable approach. Various Artificial Neural Network (ANN) models have been used to construct test oracles as they have the ability to simulate a software system's behaviour based on input/output pairs [14]. They can be used as continuous [24] or discrete [25] function approximators and that property can be exploited to build oracles. There are two operation phases in the development of an ANN: training and regression (or association, if the ANN is used as a classifier) [24]. Given a training set composed of input/output pairs, the ANN (in the role of a continuous function approximator) is capable of finding an approximate function of a deterministic computational process. In the role of a regression model, the trained ANN can generate the expected outputs to input data that are not part of the training set. The ANN used as a discrete function approximator can be trained with a set of input/output pairs, where the output is a category to input, and then classify other unseen inputs in one of the given categories (this scenario was illustrated earlier in Fig. 10.5 in Sec. 10.2).

For example, Aggarwal *et al.* [25], Chan *et al.* [26] and Jin *et al.* [24] tackle the use of ANN as oracles to the problem of test output classification.

Two of these papers present a case study for creating an oracle for the famous triangle problem. The inputs are three integers that represent the length sides of a triangle. The output is the classification into equilateral, isosceles, scalene or not a triangle. The ANN is given correct input/output pairs as a training set and after training phase is able to classify new inputs into the presented categories.

Vanmali and colleagues trained a multi-layer ANN on an original software application by using randomly generated test data that conformed to the specification [27]. When new versions of the original application are created and regression testing was required, the tested code was executed on the test data to yield outputs that are compared with those of the ANN. Shahamiri *et al.* [14] presented an experiment with a student registration verifier program which validates the registration, decides the maximum courses students can select and if a discount is applicable or not. A backpropagation ANN was used as an oracle and evaluated on the golden (reference) version test cases and mutated test cases.

All of previous studies used a single ANN oracle. Shahamiri *et al.* [28] proposed a multi-ANN oracle to perform input/output mapping in order to test more complicated software applications where a single ANN oracle may fail to deliver a high quality oracle. A single ANN was defined for each of the output items of the output domain; then all of ANN together made the oracle. As a result, the complexity of the software may be distributed between several ANN instead of having a single one to do all of the learning, and also separating the ANN may reduce the complexity of the training process and increase the oracle's ability to find faults. The experimental results indicate that multi-ANN oracle performed much better than single ANN oracle. However, building single ANN for every output item to create multi ANN oracle could be expensive.

Although all of previous studies demonstrate the ability of ANNs to act as a test oracle, they may not be reliable when the complexities of subject programs increase because they require larger training samples that could make the ANN learning process complicated. A small ANN error could increase the oracle miss-classification error considerably in large software applications. Moreover, most of these studies were evaluated by small subject programs having small input/output domains and the ANN was able to perform the mapping in most of these studies. It is possible that a tiny difference exists between expected output generated by the ANN based oracle and the correct program. These issues could happen because of the complexity of the application under test. Consequently, generating

the most representative data sets to train the ANN could enhance the ANN performance and reduce the mis-classification error. In addition, the structure of the ANN (e.g. the number of layers and neurons) is another issue which may not be easy to determine.

More recently, Tsimpourlas *et al.* [29] use supervised learning over test execution traces. A small fraction of the execution traces were labelled with their verdict of pass or fail, and then used to train an ANN model to learn and distinguish run-time patterns for passing versus failing executions for a given program. Their experimental results showed that the classification model was highly effective in classifying passing and failing executions, achieving over 95% precision, recall and specificity while only training with an average 9% of the total traces.

ANN algorithms are not the only supervised learning models used. Wang *et al.* [30] applied support vector machine (SVM) as a supervised learning algorithm to test reactive systems. Parsa *et al.* [31] trained a support vector machine (SVM) to detect faults during the execution of subject programs. Their work was extended by using a SVM with a customised kernel function to measure the similarities between passing and failing executions, represented as sequences of program predicates [32]. Frounchi *et al.* [33] used a decision tree technique as a test oracle to verify the accuracy of an image segmentation algorithm which was able to achieve an average accuracy of approximately 90% during the evaluation phase. Brun and Ernst [34] also explored the use of SVM and decision tree to rank program properties provided by the user that are likely to indicate errors in the program.

Other learning algorithms were used by Haran *et al.* [35] to classify execution data collected from applications in the field as coming from either passing or failing program runs. They used random forests to model and predict the outcome of an execution based on the corresponding execution data. Their work was extended by proposing two different classification techniques (association trees and adaptive sampling association trees) which can build models with significantly less data than that required by random forests but maintaining the same accuracy [36]. Lo *et al.* [37] proposed a new technique to classify unknown executions. Their technique first mined a set of discriminative features capturing repetitive series of events from program execution traces. After that, feature selection was performed in order to select the best features for classification. Then, these features were used to train a classifier (SVM) to detect failures.

10.4.2 *Test Oracles Based on Semi-Supervised Learning Techniques*

Semi-supervised learning techniques are employed in situations where labelled data is scarce (e.g. instances may be difficult to come by or expensive to label). This make them very appropriate for the test oracle problem, where there may be a small set of labelled results (i.e. classified by a human as passing or failing — normal or abnormal) which may then be used to incrementally create more sophisticated classifiers.

Some of the related work in this subsection combines supervised learning with unsupervised learning, but we consider them as semi-supervised learning techniques. For instance, clustering is often performed as a preliminary step in the data mining process with the resulting clusters being used as further inputs into downstream techniques such as a neural network (it is often helpful to apply clustering analysis first to reduce the search space for the downstream algorithm). Podgurski *et al.* built a system to cluster bugs represented by a failed test that had the same cause [38]. Their approach was based on the analysis of the execution profile corresponding to reported failures of the test and was built on top of their earlier unsupervised learning system where the execution count for each function in the program was used as a feature to construct the model. Francis *et al.* proposed two new tree-based techniques for refining an initial classification of failures [39]. The first of these techniques was based on the use of dendrograms which are tree-like diagrams used to represent the results of hierarchical cluster analysis. Their dendrogram-based technique for refining failure classification was used to decide how non-homogeneous clusters should be considered for merging. The second technique for refining an initial failure classification relied on generating a classification tree to recognise failed executions. A classification tree was constructed algorithmically using a training set containing positive and negative instances of the class of interest. The experimental results indicated that both techniques were effective for grouping together failures with the same or similar causes. All techniques were aimed at bug localisation by identifying groups of failures with closely related causes among a set of reported failures based on user feedback.

Bowring and colleagues proposed an automatic classification of program behaviours using execution data aimed at reverse engineering a more abstract description of system's behaviour [40]. Their work focused on an active learning approach (rather than batch learning approach) where, for

each iteration of learning, the classifier is trained incrementally on a series of labelled data elements and then applied to series of unlabelled data to predict those elements that most significantly extend the range of behaviours that can be classified. These selected elements are then labelled and added to the training set for the next round of learning. Their technique builds a classifier for software behaviour in two stages. Initially, a model of individual program executions was built as a Markov model by using the profiles of event transitions such as branches (a binary matrix was used to transform data to a suitable set of feature vectors). Each of these models thus represents one instance of the program's behaviour. The technique then used an automatic clustering algorithm to build clusters of these Markov models, which then together form a classifier tuned to predict specific behavioural characteristics of the considered program. Mao *et al.* [41] also used the Markov model approach proposed by Bowring *et al.* [40] and clustering analysis along with a new sampling strategy (priority-ranked n-per-cluster) to aid fault localisation. The methodology starts by using a Markov model and the profiles of event transitions such as branches to depict program behaviours. Based on the obtained model, the dissimilarity of two profiles is defined. After separating the failure executions and non-failure executions into different subsets, the clustering and sampling strategy were performed on the failure execution subset in order to choose the most representative sample of failures to reduce the debugging effort.

Baah *et al.* proposed a new machine learning technique that performs anomaly detection during software execution [42]. A Markov model was trained on trace predicate information and the Baum-Welch algorithm was used to find unknown parameters for the Markov model. Probes were inserted into the subject program to sample tuples in the form of <class name, method name, line number, predicate state>. Clustering of predicate states was used also in the training phase to gather predicate state information based on the line number and method number. In the line number clustering, all predicate states generated at an instrumented line number are grouped into one cluster. In method clustering, all predicate states belonging to a method are grouped into one cluster. The subject program along with the Markov model were then deployed together to detect faults as they occur and to possibly perform fault correction actions to prevent failures. The experimental results showed that the proposed technique performed well with domain faults with up to 100% accuracy in some cases. However, the technique did not perform well with computation errors with accuracy less than 50% and dropping to 0% in some cases. The

authors pointed out to a few efficiency issues such as the time required to build such model especially in the presence of a large test suite, the cost of instrumenting the software to gather more information without incurring a significant overhead, and how quickly the model can track the execution of the software.

Semi-supervised learning techniques have been used as test oracles to classify passing and failing tests [43]. A learner is built by using training data set that has a small subset of labelled test data which is then used to classify the remaining data (i.e. labelling it as a passing or failing test). Different learning algorithms were explored on three systems based on dynamic execution data (firstly input/output pairs alone, and then input/output pairs combined with their corresponding execution traces). Two labelling strategies were used for the training data (labelled instances for both failing and passing tests, and just for passing tests alone). The experimental results showed that the proposed approach has an important practical implication: testers need to examine a small subset of the test results from a system and then use this information to train a learning algorithm to classify the rest. Roper [44] combined unsupervised and semi-supervised learning strategies together to construct test oracles by using the outcomes of applying unsupervised learning techniques as input to the semi-supervised learning techniques. The test classification strategy consists of two phases: Firstly, unsupervised learning (clustering) is used with the aim of creating a grouping of tests where the smallest clusters contain a greater proportion of failures. Manual checking of tests then focuses on these smallest clusters first as they are more likely to contain failing tests. Secondly, having checked a small proportion of the test outcomes, semi-supervised learning is then employed to use this information to label an initial small set of data and derive an automatic pass/fail classification for the remainder of tests. The combined effect of these is to create a far more efficient process than just checking the outcome of every test in order: clustering creates a small subset of tests in which failures are more prevalent, and using semi-supervised learning allows the tester to focus next on those outputs considered to be failures. The scenario for a test oracle based on semi-supervised learning techniques is shown in Fig. 10.8. This scenario is different compared to other scenarios in Sec. 10.2 as here the tester provides the semi-supervised learning techniques with some information to improve the oracle.

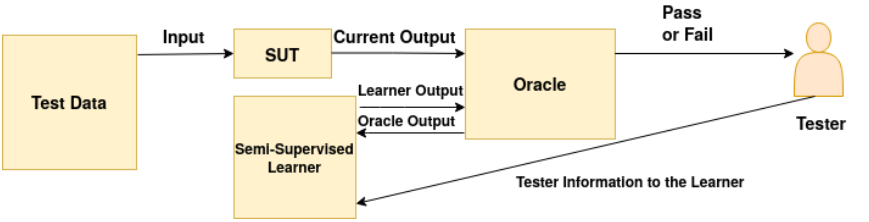


Fig. 10.8 Test Oracle Structure Using Test Data and Based on Semi-Supervised Learning Techniques.

10.4.3 *Test Oracles Based on Unsupervised Learning Techniques*

Unsupervised learning techniques do not require training data, and thus are most widely applicable. The techniques based on unsupervised learning make the implicit hypothesis that abnormal instances are relatively infrequent in the test data compared to normal instances. If this hypothesis is not true, then the techniques will suffer from high false positive rate. Dickinson, Leon and Podgurski demonstrated the advantage of automated clustering of execution profiles over random selection for finding failures by using function caller/callee feature profiles as the basis for cluster formation [45, 46]. This work is in turn based on that of Podgurski *et al.* [47], who used cluster analysis of profiles (the execution counts of conditional branches) and stratified random sampling to calculate estimates of software reliability, and found that failures were often isolated in small clusters based on unusual execution profiles.

Yoo *et al.* applied clustering to the problem of regression test optimisation [48] where test cases are clustered based on their dynamic runtime behaviour (execution traces). Their experimental results showed that the clustering approach outperformed the coverage based approach in terms of fault detection rate. Yan *et al.* [49] proposed a dynamic test cluster sampling strategy called execution spectra based sampling (ESBS). The empirical evaluation showed that the proposed sampling strategy is more effective than existing test cluster sampling strategies [45, 46]. Masri *et al.* [50] presented an empirical study of several test case filtering techniques (coverage based and distribution based techniques) by using various types of information flows (e.g. basic blocks, branches, function calls and call pairs). Their empirical study showed that coverage maximization and distribution-based filtering techniques were more effective overall than simple random

sampling. In addition, distribution-based filtering techniques did not perform significantly better than coverage maximization overall.

More recently, Almaghairbe and Roper have investigated the use of clustering based anomaly detection techniques to support the construction of a test oracle by performing two different empirical studies along with varying types of dynamic execution data [51, 52]. In the first study, a range of clustering algorithms were applied to just the input-output pairs of three systems with the primary aim of exploring the feasibility of this approach. The aim of the second study was to improve the accuracy and performance of the approach by augmenting the input/output pairs with their associated execution traces. Their results demonstrate important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining a relatively small proportion of the data to discover a large proportion of the failures. Figure 10.9 shows the test oracle structure based on unsupervised learning techniques. It can be observed that the approach is built based on the SUT output only (no other information) to judge suspicious outputs. Table 10.1 summarises the difference between test oracle structures presented in this chapter.

Fig. 10.9 Test Oracle Structure Using Test Data and Based on Unsupervised Learning Techniques.

10.4.4 *Summary and Findings*

Returning to the research questions in Sec. 10.1, the principal findings of this chapter may be summarised as follows:

- Question 1:
  (a) The techniques that can be used to build an automated test oracle are divided into three categories based on the learning strategies:
    i. Unsupervised machine learning techniques such as clustering methods with different sampling strategies (stratified, simple random, one-per-cluster, adaptive, n-per-cluster and failure-pursuit sampling strategies) or without any sampling strategies.

Table 10.1 The Summary of Different Concepts of Test Oracles.

Test Oracles Structures	
Test Oracles	The Different
Test oracles based on individual test cases (Fig. 10.2 and Fig. 10.3 in Sec. 10.2)	This scenario requires a tool to generate the expected output (e.g. JUnit) or tester knowledge about the SUT.
Test oracles based on formal specifications (Fig. 10.4 and Fig. 10.5 in Sec. 10.2)	This scenario requires expected output generator such as a formal method or ANN.
Test oracles based on semi-supervised learning techniques (Fig. 10.8 in Sec. 10.4)	The tester in this scenario provides semi-supervised learning techniques with some information to improve the oracle.
Test oracles based on unsupervised learning techniques (Fig. 10.9 in Sec. 10.4)	The approach in this scenario is built based on the SUT output only (no other information) to judge suspicious outputs.

ii. Semi-supervised learning techniques such as logistic regression with clustering methods and Markov model with clustering methods. Traditional semi-supervised learning techniques have been also used such as self-training and co-training methods. The results of clustering methods have also been used as input to semi-supervised/supervised learning techniques.

iii. Supervised machine learning techniques such as a multi-layered perceptron neural network, back-propagation neural network, radial basis function neural network (RBF), support vector machine (SVM) with kernel function, decision tree (DT), random forests, association trees, adaptive sampling association trees, and frequent pattern mining algorithm.

Figure 10.10 illustrates the distribution of the learning strategies of the studies discussed by counting the number of studies using each learning strategy.

(b) The chapter reported that the range of anomaly detection approaches explored were applicable for automated test oracles but different machine learning techniques with different dynamic execution data give mixed results in terms of detection accuracy. However, some techniques have been shown to be superior in term of their software fault detection accuracy. These are presented based on their learning strategy as follows:

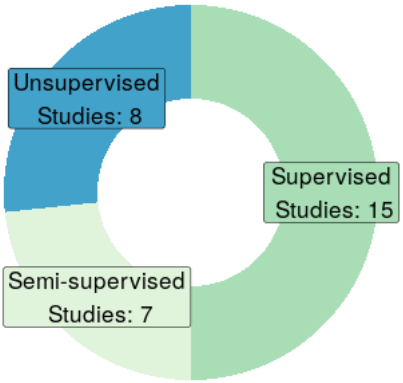


Fig. 10.10 Learning Strategies Used in the Studies.

i. Supervised machine learning techniques: Artificial neural network algorithms (ANN) showed significant accuracy in term of software fault detection for the oracle problem by using only a set of input/output pairs of SUT as feature. In addition, decision tree (DT) performed much better than support vector machine (SVM) in terms of classification accuracy by using program properties as feature. However, support vector machine (SVM) with a kernel function showed reasonable accuracy in software fault detection by using predicate state information of the program as feature. Moreover, each of random forests, association trees and adaptive sampling association trees also showed reasonable accuracy in term of software behaviour classification by using different execution data. Furthermore, frequently the pattern mining algorithm also performed well for software fault detection and localisation by using execution traces.

ii. Semi-supervised machine learning techniques: Logistic regression and clustering techniques performed well when used in combination to group failures that have the same cause together by using execution profile as the feature set. In addition, Markov model with clustering techniques performed well when used together to classify software behaviour by using both profile event transitions and predicate state information as a set of features. Furthermore, traditional semi-supervised learning techniques demonstrates their ability to compete alongside supervised learning

techniques in terms of constructing an automated test oracles. In addition, the combination of clustering algorithms and semi-supervised/supervised learning techniques also showed reasonable performance.

iii. Unsupervised machine learning techniques: Clustering methods with different sampling strategies (stratified, simple random, one-per-cluster, adaptive, n-per-cluster and failure-pursuit sampling strategies) showed their ability to find failures by using different execution profiles. Simple random sampling strategy did not perform well compared to other sampling strategies. In addition, clustering algorithms with no sampling strategies proved that they can be used to build automated test oracles.

The number of individual machine learning algorithms that has been used in discussed papers is illustrated in Fig. 10.11 (fifteen Algorithms in total). From the algorithms shown in Fig. 10.9, the five most frequently individual algorithms are ANN, Hierarchical clustering with sample strategies, SVM, Markov model with clustering and DT. ANN is the most frequently used individual algorithm for test oracle construction with seven studies.

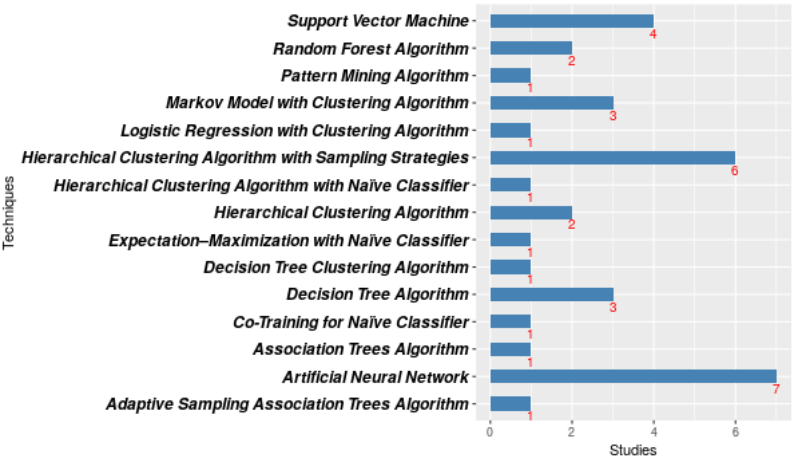


Fig. 10.11 Individual Algorithms Used for Test Oracle Construction.

(c) In terms of effectiveness, a classification strategy (supervised machine learning techniques) is reported to be better in most cases compared

to a clustering strategy (unsupervised machine learning techniques) in terms of software fault detection. However, a classification strategy has disadvantages that can be summarised as follows (Chandola *et al.*, 2009): (1) multi-class classification-based techniques rely on the availability of accurate labels for various normal classes, which is often not possible; (2) classification-based techniques assign a label to each test instance, which can also become a disadvantage when a meaningful anomaly score is desired for the test instances.

As can be seen from Fig. 10.12, classification problems used far more frequently than other machine learning problems when constructing test oracles.

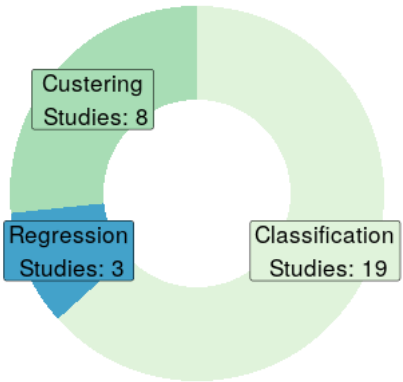


Fig. 10.12 The Frequency of Machine Learning Problem Types Used to Construct Test Oracles.

- Question 2:

  (a) The chapter reported that different types of dynamic execution data have been used as a set of features to build anomaly detection model such as the execution count of conditional branches, function caller/callee profiles, execution count for functions or methods, input/output pairs, throw counts, catch counts and execution traces such as temporal relation events, program properties, frequent path sequences, profiles of event transitions, predicate state information, and method entry/exit points.
  Figure 10.13 shows the frequency of execution data usage in discussed papers. Input/Output pairs and execution profiles are the most

Fig. 10.13 Frequency of Use of Execution Data Types Over the Studies.

frequently used feature for building test oracles (9 studies each), followed by execution traces, combinations of input/output pairs and execution traces, decision control statements and profiles of event transition (3 and 2 studies for each feature respectively). Predicate state information is only used in 1 study.

(b) The experimental results in some studies showed that the execution count for conditional branches and execution count for functions or methods are more suitable to build an effective automated software fault detection model (using a supervised learning strategy) compared to throw count and catch count. In addition, the combination of input/output pairs and execution traces is more suitable to build an effective test oracle compared to input/output pairs alone.

(c) The chapter reported that different approaches have been used to transform dynamic execution data to suitable set of feature vectors for anomaly detection techniques to be able to make meaningful distance comparisons between execution trace profiles such as binary metric, proportional metric, SD metric, histogram metric, linear regression metric, count binary metric and proportional binary metric [45]. However, binary metric is the most commonly used approach, and normalisation has only been used in the case of using input/output data with anomaly detection to build an automated test oracle. Tokenisation and compression is also used to transform

text input/output pairs and execution traces into suitable and compact feature representations.

- Question 3:

  (a) The chapter reported different types of faults that have been used in the experiments such as operator faults, domain faults, computation faults, fatal faults (executions that terminated because an uncaught exception), non-fatal faults (executions that terminated normally but produce the wrong output), omission faults (omission bugs are those where methods/functions that should have been invoked were absent), additional faults (bugs are those where methods/functions were invoked but were unnecessary and caused a failure of execution run) and ordering faults (bugs are those methods/functions that are called out of sequence).

  (b) The experimental results for some studies have reported that domain faults are more frequently detected by the Markov model compared to computation errors, and also omission and additional bugs are more frequently detected by the pattern mining algorithm compared to ordering bugs.

  (c) There is no evidence for any relationship between specific types of faults and anomaly detection approaches that have been reported in discussed papers in this chapter.

Tables 10.2–10.4 provide a summary of the available approaches, the underlying mechanisms and input/output data. In addition, Tables 10.5–10.7 highlight the main findings for all proposed approaches in the paper discussed in this chapter.

10.5 Discussion

In terms of their application in practice, the most important properties that test oracles need to demonstrate are scalability, fault detection ability, a low false positive rate and cost effectiveness. Each of those properties is explained further below:

- Scalability means the ability of any technique to handle any size of software (with corresponding increases in the volume of data). In other words, a technique has to be potentially usable at an industrial level.

Table 10.2 Summary of Selected Studies Using Supervised Learning.

Techniques Based on Supervised Learning Strategy				
Author	Aim	Technique	Data	Transformation - Processing
Jin *et al.* (2008)	Test oracle	ANN	Input/Output pairs	N/A
Aggarwal *et al.* (2004)	Test oracle	ANN	Input/Output pairs	N/A
Chan *et al.* (2006)	Test oracle	ANN	Input/Output pairs	Normalisation
Vanmali *et al.* (2002)	Test oracle	ANN	Input/Output pairs	Normalisation - Binary
Shahamiri *et al.* (2010)	Test oracle	ANN	Input/Output pairs	Normalisation - Binary
Shahamiri *et al.* (2012)	Test oracle	ANN	Input/Output pairs	Normalisation - Binary
Tsimpourlas *et al.* (2020)	Test oracle	ANN	Execution traces (sequences of method invocations)	Binary
Wang *et al.* (2011)	Test oracle	SVM	Execution traces (variables for temporal Relation events)	Binary
Parsa *et al.* (2009)	Faults detection	SVM	Decision control statements	Binary
Parsa *et al.* (2012)	Faults detection	SVM	Decision control statements	Binary
Frounchi *et al.* (2011)	Test oracle	DT	Input/Output pairs	Normalisation
Brun *et al.* (2004)	Finding error in program properties	SVM and DT	Execution traces (program properties)	Binary
Haran *et al.* (2005)	Classifying execution data	Random forests and DT	Statement, throw/catch and method counts	Principal component analysis
Haran *et al.* (2007)	Classifying execution data	Random forests, association trees and adaptive sampling association trees	statement, throw/catch and method counts	Principal component analysis
Lo *et al.* (2009)	Failures detection	Pattern mining algorithm	Execution traces (the frequent path sequences with timing information)	Binary

- Fault detection ability refers to the effectiveness which new (unseen) faults occurring in the running application are identified.
- False positive rate is the rate of false alarms reported by test oracles. This can be considered as the biggest issue with automated oracles. When such a rate is intolerably high, any problem reported by automated oracles will be deemed unreliable and ignored by developers.

Table 10.3 Summary of Selected Studies Using Semi-Supervised Learning.

Techniques Based on Semi-Supervised Learning Strategy				
Author	Aim	Technique	Data	Transformation - Processing
Podgurski *et al.* (2003)	Built a system to group together failures with the same or similar causes	Logistic regression and clustering technique	Execution function counts	Binary
Francis *et al.* (2004)	Built a system to classifying reported instances of software failures so that failures with the same cause are grouped together	tree-based techniques (e.g. CART algorithm) and clustering technique	Execution function/method counts	Binary
Bowring *et al.* (2004)	Automatic classification of program behaviours using execution data	Markov model and clustering technique	Profiles of event transitions such as branches	Binary
Mao *et al.* (2005)	Software faults localization	Markov model and clustering technique with priority-ranked-n-per-cluster sampling strategy	Profiles of event transitions such as branches	Binary
Baah *et al.* (2006)	Software faults detection and localization on deployment stage	Markov model with clustering technique	Predicate state information such as class name, method name, line number and predicate state	Binary
Almaghairbe *et al.* (2016)	Test oracle	Self-training and co-training techniques	Input/Output pairs	Tokenization process
Roper (2019)	Test oracle	Cluster analysis with supervised learning techniques	Input/Output pairs and execution traces (methods entry/exit points)	Tokenization process for input/output pairs and hash function for execution trace

- Cost effectiveness takes into consideration the effort and resources required to create an oracle in relation to its ability to reveal subtle semantic failures.

Generally, all those properties are complementary to each other and can affect the usability of any test oracle in practice. The ultimate goal of the software testing community is to find a test oracle that can be used to test any system, and is able to find all failures with a low false positive rate at an acceptable cost.

Test oracles based on supervised learning techniques have been widely used to build an automated test oracle. They have shown that they are

Table 10.4 Summary of Selected Studies Using Unsupervised Learning.

Techniques Based on Unsupervised Learning Strategy				
Author	Aim	Technique	Data	Transformation - Processing
Podgurski *et al.* (1999)	Estimation of software reliability	Cluster analysis with stratified sampling strategy	Execution profiles (execution counts of conditional branches)	Binary
Dickinson *et al.* (2001a)	Finding software failures	Cluster analysis with different sampling strategies (simple random, one-per-cluster, adaptive and n-per-cluster)	Execution profiles (function caller/callee)	Binary, proportional, histogram, standard deviation, linear regression, count binary and proportional count binary
Dickinson *et al.* (2001b)	Finding software failures	Cluster analysis with different sampling strategies (failure-pursuit, simple random, one-per-cluster, adaptive and n-per-cluster)	Execution profiles (function caller/callee)	Binary, proportional, histogram, standard deviation, linear regression, count binary and proportional count binary
Yoo *et al.* (2009)	Finding software failures	Cluster analysis without sampling strategy	Execution profiles (execution counts of conditional branches)	Binary
Yan *et al.* (2010)	Finding software failures	Cluster analysis with execution-spectra-based sampling strategy (ESBS)	Execution profiles (execution counts of conditional branches)	Binary
Masri *et al.* (2007)	Finding software failures	Cluster analysis via coverage based and distribution based filtering techniques	Execution profiles (basic blocks, branches, function calls and call pairs)	Numeric
Almaghairbe *et al.* (2015)	Test oracle	Cluster analysis without sampling strategy	Input/Output pairs	Tokenization process
Almaghairbe *et al.* (2017)	Test oracle	Cluster analysis without sampling strategy	Input/Output pairs and execution traces (methods entry/exit points)	Tokenization process for input/output pairs and hash function for execution trace

able to test any system with any size which make them scalable. They also tend to display a powerful ability to detect failures with a very low false positive and false negative rate (in other words, a high classification accuracy). However, their effectiveness depends on the availability of a fully labelled training data set (each instance in the training data set has to be labelled as passing or failing test execution) to construct the oracle. Labelling each instance in the training data set can be an expensive process and typically relies on using a reference version of the software (which is difficult to obtain in practice), making them prohibitively expensive and not cost-effective.

Table 10.5 Summary of Findings for Selected Studies Using Supervised Learning.

Techniques Based on Supervised Learning strategy	
Author	Findings
Jin *et al.* (2008)	The average accuracy for the technique is approximately 60% over several data sets.
Aggarwal *et al.* (2004)	The mean miss-classification error for the technique is 15.9% with standard deviation of 2.312. The mean non-prediction error for the technique is only 2.949% with standard deviation of 1.252. The total mean relative error is found to be 19.02% with standard deviation of 2.224.
Chan *et al.* (2006)	The average accuracy for the technique is 78.14%.
Vanmali *et al.* (2002)	The minimum classification error rate for the technique for the binary output is 8.31% and for the continuous output is 20.79%.
Shahamiri *et al.* (2010)	The average accuracy for the technique is 95.37%.
Shahamiri *et al.* (2012)	The average accuracy for the technique in the first case study is 95.7% and for the second case study is 98.83%.
Tsimpourlas *et al.* (2020)	The classification model for each of the subject programs is highly effective in classifying passing and failing executions, achieving over 95% precision, recall and specificity while only training with an average 9% of the total traces.
Wang *et al.* (2011)	The average accuracy for the technique is 95.46%.
Parsa *et al.* (2009)	The average precision for the technique is 85%.
Parsa *et al.* (2012)	The average precision for technique is 65%.
Frounchi *et al.* (2011)	The average accuracy for the technique is 95%.
Brun *et al.* (2004)	The experimental results showed for C programs, 45% of the top 80 properties are fault-revealing. For Java programs, 59% of the top 80 properties are fault-revealing. The DT technique performed much better than SVM in term of classification. The SVM technique performed much better than DT technique in term of ranking properties.
Haran *et al.* (2005)	Statement counts and method counts succeeded in building classification model with higher accuracy compared to the classification model used throw counts and catch counts.
Haran *et al.* (2007)	Three techniques performed well with overall miss-classification rates typically below 2 percent.
Lo *et al.* (2009)	The average accuracy for proposed algorithm was 93.77% and 24.67% over the base classifier. The technique worked well with omission and additional bugs but poorly with ordering bugs.

Table 10.6 Summary of Findings for Selected Studies Using Semi-Supervised Learning.

Techniques Based on Semi-Supervised Learning Strategy	
Author	Findings
Podgurski *et al.* (2003)	These results indicate that the strategy can be effective and is able to group together failures with the same or similar causes.
Francis *et al.* (2004)	Experimental results indicate that the proposed techniques are effective for grouping together failures with the same or similar causes.
Bowring *et al.* (2004)	An active learning approach had very high classification accuracy and performed better than batch learning approach.
Mao *et al.* (2005)	The results show that the clustering and sampling techniques based on revised Markov model is more effective to find faults than Podgurski's method (one-per-cluster sampling method).
Baah *et al.* (2006)	The technique performed well with domain faults and performed poorly with computation error.
Almaghairbe *et al.* (2016)	The results show that in many cases labelling just a small proportion of the test cases as low as 10% is sufficient to build a classifier that is able to correctly categorise the large majority of the remaining test cases.
Roper (2019)	The accuracy of the technique is 86% with just only 31 test cases labelled.

Test oracles based on semi-supervised learning techniques are less expensive in comparison to those based on supervised learning techniques as they require a smaller set of labelled training data (as opposed to the large data set required by supervised techniques or the fault-free version employed by invariant detectors). However, test oracles based on semi-supervised learning techniques have a lower accuracy in comparison to those based on supervised learning techniques (they have a slightly higher false positive rate, and also slightly lower fault detection ability) which is to be expected as the training of the algorithms use far less labelled data. Semi-supervised approaches are a classic demonstration of the cost benefit trade-off: a larger set of labelled data is likely to yield a more accurate classifier, and while these techniques are significantly more cost-effective (and practicable) than supervised approaches, there is still work to be done in establishing the ideal ratio of labelled to unlabelled data.

Test oracles based on unsupervised learning techniques do not require the availability of labelled data or a fault free version of the SUT to construct test oracles which make them more scalable in comparison to test oracles based on supervised/semi-supervised learning techniques and test oracles based on invariant detection in terms of the provision of labelled data (other scalability issues may arise in the application of the algorithms

Table 10.7 Summary of Findings for Selected Studies Using Unsupervised Learning.

Techniques Based on Unsupervised Learning Strategy	
Author	Findings
Podgurski *et al.* (1999)	Experimental results suggest that the approach is computationally feasible, can isolate failures, and can significantly reduce the cost of estimating software reliability. Stratified sampling strategy performed much better than simple random sampling strategy.
Dickinson *et al.* (2001a)	The experimental data shows that the percentage of failures found in the smallest clusters is significantly higher than 50%. One-per-cluster, adaptive and n-per-cluster sampling strategies performed much better than simple random sampling strategy. Also, binary, standard deviation, histogram, proportional and proportional binary metrics with the sampling strategies performed much better than linear regression and binary metrics.
Dickinson *et al.* (2001b)	The experimental data shows that the percentage of failures found in the smallest clusters is significantly higher than 50%. Failure-pursuit, one-per-cluster, adaptive and n-per-cluster sampling strategies performed much better than simple random sampling strategy. Also, binary, standard deviation, histogram, proportional and proportional binary metrics with the sampling strategies performed much better than linear regression and binary metrics.
Yoo *et al.* (2009)	The empirical studies show that the proposed approach (hybrid ICP algorithm) can outperform the traditional coverage-based prioritisation for some programs.
Yan *et al.* (2010)	The experimental results show that clustering algorithm with ESBS sampling strategy is better in failures detection than existing sampling strategies in most case.
Masri *et al.* (2007)	Both coverage maximization and distribution-based filtering techniques were more effective overall than simple random sampling, although the latter performed well in one case in which failures comprised a relatively large proportion of the test suite. In addition, distribution based filtering techniques did not perform significantly better than coverage maximization overall.
Almaghairbe *et al.* (2015)	The findings reveal that failing outputs do indeed tend to congregate in small clusters, suggesting that the approach is feasible and has the potential to reduce by an order of magnitude the numbers of outputs that would need to be manually examined following a test run.
Almaghairbe *et al.* (2017)	The experimental results gave an evidence which support the clustering hypothesis behind their work where in several cases small (less than average sized) clusters contained more than 60% of failures (and often a substantially higher proportion). As well as having a higher failure density they also contained a spread of failures in the cases where there were multiple faults in the programs. The results also demonstrate important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining a relatively small proportion of the data to discover a large proportion of the failures.

but these are likely to be equally applicable to all approaches). In addition, they are less expensive to obtain in comparison to test oracles based on supervised/semi-supervised learning techniques (again as no data labelling is necessary), but can be less accurate for the same reason.

Most recently, Almaghairbe *et al.* [53, 54] empirically evaluated test oracles based on semi-supervised and unsupervised learning techniques by comparing them with existing techniques from the mining specification domain (the data invariant detector Daikon [55]). The experimental results showed that semi-supervised learning techniques have a higher fault detection ability (82% on average) and lower false positive rate (11% on average

compared to 16% for Daikon and for unsupervised learning). In addition, unsupervised learning techniques have slightly higher fault detection ability compared to Daikon (66% compared to 64%), but a similar false positive rate (16% on average).

In general, as such approaches are developed, more work is needed on the measurement of oracles and their properties, and also it has been suggested that it is important for the software metrics community to consider the concept of "oracle metrics" [4].

10.6 Further Research Direction

Despite the achievements identified in this chapter, there are a number of barriers that need to be overcome for the work to become practically usable which fall into the categories of scalability and accuracy. These are explained further below:

10.6.1 *Improving the Accuracy*

Accuracy in this context means the ability of the proposed approaches to identify failing and passing test results as correctly as possible (high true positive rate and low false positive/false negative rates). The accuracy of semi-supervised and unsupervised learning techniques may be improved by augmenting the data sets (input/output pairs and execution traces) with more relevant information from the program execution (e.g. state information, execution time and code coverage etc.) to build more effective/accurate test oracles, but this strategy still needs to be evaluated.

Adding more execution data to the data sets can help to reduce the size of labelled data needed to train the learning algorithms in semi-supervised learning. This also relates to scalability but to make the approach practical the size of labelled data needs to be as low as possible which means improving the accuracy as well. The other point related to the size of labelled data and then accuracy is that the predictions of semi-supervised learning approaches should come with an estimate of confidence.

The accuracy of unsupervised learning approaches can be improved by selecting the most appropriate similarity measures for clustering algorithms. In addition, the number of specified clusters for clustering algorithms is important and the accuracy can be improved by specifying the optimal number of cluster counts as this can have a significant impact on the way that passing and failing tests become grouped into different clusters.

The final way to improve accuracy is to investigate alternative algorithms. A wide range have already been employed as has been seen in this study but there are many machine learning algorithms which have yet to be explored, along with meta-level approaches such as ensemble methods which combine learners to build more effective and accurate algorithms.

10.6.2 *Improving the Scalability*

Many of the approaches reviewed work with very complex feature sets: test inputs and outputs, execution traces, function call profiles, predicate state information, event transition profiles etc. These are often extensive sets of information which are not easily handled by machine learning algorithms and consequently need to be transformed and encoded in a form that allows them to be compared sensibly and used meaningfully as feature sets. For instance, the input/output pairs for tested systems in [43, 51, 52] were of string/text type and it turned out that a tokenization procedure worked reasonably well with the proposed approach, but this may not be generally applicable for all input and output types. More general strategies need to be identified to make the approaches scalable and widely applicable.

An interesting approach to addressing the scalability (and cost-effectiveness) issue which should be explored in the future is to investigate the feasibility of using the cheap results from clustering in which there is the greatest confidence to generate the labelled set required by the semi-supervised techniques, and thereby reduce the cost of the semi-supervised learning. There has already been some initial work in this area [44]

In summary, the fundamental principle to the successful automated test oracles is the capability to build oracles that demonstrate a substantially lower false positive rate and higher fault detection capability, as compared to the available, state of the art tools. Therefore, future research is also devoted to further development and empirical investigation of the effectiveness of several automated test oracles, to evaluate the features offered by different alternative oracles.

10.7 Conclusion

The importance of testing activity is widely known. However, the difficulty in deciding whether a result is acceptable or not (also known as the oracle problem) hampers such activity.

The comparison between expected output and obtained output is often

determined manually by the tester, which is usually slow, error-prone, and very expensive. Several approaches have been proposed in order to address the oracle problem and automate the process.

The main contribution of this chapter is to present a comprehensive overview on test oracles based on machine learning techniques, and identify the strengths and limitations of the approach. We also provided a discussion about the required properties of automated and semi-automated test oracle techniques (based on machine learning techniques) for them to be practically usable such as scalability, accuracy and cost effectiveness. Those properties are used as criteria to evaluate the existing approaches. To conclude, the chapter sets out a road map for future work and a guideline to software testing researchers who seek to address this challenging and important problem.

References

[1] G. Fraser and A. Arcuri, Evosuite: Automatic test suite generation for object-oriented software, in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11. ACM, New York, NY, USA, ISBN 978-1-4503-0443-6, pp. 416–419 (2011), ISBN 978-1-4503-0443-6, doi:10.1145/2025113.2025179, `http://doi.acm.org/10.1145/2025113.2025179`.

[2] L. Baresi and M. Young, Test oracles, Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A. (2001), `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

[3] C. Pacheco and M. D. Ernst, Randoop: Feedback-directed random testing for java, in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07. ACM, New York, NY, USA, ISBN 978-1-59593-865-7, pp. 815–816 (2007), ISBN 978-1-59593-865-7, doi:10.1145/1297846.1297902, `http://doi.acm.org/10.1145/1297846.1297902`.

[4] E. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, The oracle problem in software testing: A survey, *Software Engineering, IEEE Transactions on* **41**, 5, pp. 507–525 (2015), doi:10.1109/TSE.2014.2372785.

[5] D. Tu, R. Chen, Z. Du and Y. Liu, A method of log file analysis for test oracle, in *Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on*, pp. 351–354 (2009), doi:10.1109/EmbeddedCom-ScalCom.2009.69.

[6] A. Memon, I. Banerjee and A. Nagarajan, What test oracle should I use for effective gui testing? in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 164–173 (2003).

[7] S. R. Shahamiri, W. M. N. W. Kadir and S. Z. Mohd-Hashim, A comparative study on automated software test oracle methods, in *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pp. 140–145 (2009), doi:10.1109/ICSEA.2009.29.

[8] M. Ye, B. Feng, L. Zhu and Y. Lin, Automated test oracle based on neural networks, in *2006 5th IEEE International Conference on Cognitive Informatics*, Vol. 1, pp. 517–522 (2006), doi:10.1109/COGINF.2006.365539.

[9] R. A. P. Oliveira, U. Kanewala and P. A. Nardi, Automated Test Oracles: State of the Art, Taxonomies, and Trends, ELSEVIER ACADEMIC PRESS INC 525 B STREET, SUITE 1900, SAN DIEGO, CA 92101-4495 USA, pp. 113–199 (2014).

[10] P. McMinn, M. Shahbaz and M. Stevenson, Search-based test input generation for string data types using the results of web queries, in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 141–150 (2012), doi:10.1109/ICST.2012.94.

[11] A. Z. Javed, P. A. Strooper and G. N. Watson, Automated generation of test cases using model-driven architecture, in *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07. IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2971-2, pp. 3– (2007), ISBN 0-7695-2971-2, doi:10.1109/AST.2007.2, `http://dx.doi.org/10.1109/AST.2007.2`.

[12] D. Hook and D. Kelly, Testing for trustworthiness in scientific software, in *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pp. 59–64 (2009), doi:10.1109/SECSE.2009.5069163.

[13] D. D'Souza and M. Gopinatha, Computing complete test graphs for hierarchical systems, in *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pp. 70–79 (2006), doi:10.1109/SEFM.2006.13.

[14] S. R. Shahamiri, W. M. N. W. Kadir and S. Ibrahim, A single-network ann-based oracle to verify logical software modules, in *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, Vol. 2, pp. V2–272–V2–276 (2010), doi:10.1109/ICSTE.2010.5608808.

[15] M.-C. Gaudel, Testing can be formal, too, in *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95. Springer-Verlag, London, UK, UK, ISBN 3-540-59293-8, pp. 82–96 (1995), ISBN 3-540-59293-8, `http://dl.acm.org/citation.cfm?id=646619.697560`.

[16] M. E. Delamaro, F. de Lourdes dos Santos Nunes and R. A. P. de Oliveira, Using concepts of content-based image retrieval to implement graphical testing oracles, *Software Testing, Verification and Reliability* **23**, 3, pp. 171–198 (2013), doi:10.1002/stvr.463, `http://dx.doi.org/10.1002/stvr.463`.

[17] A. M. Memon and Q. Xie, Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software, in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 8–17 (2004), doi:10.1109/ICSM.2004.1357785.

[18] M. Pezzè and C. Zhang, Automated test oracles: A survey, in *Advances in Computers*, Vol. 95. Elsevier, pp. 1–48 (2015).

[19] R.-K. Doong and P. G. Frankl, The astoot approach to testing object-oriented programs, *ACM Trans. Softw. Eng. Methodol.* **3**, 2, pp. 101–130 (1994), doi:10.1145/192218.192221, http://doi.acm.org/10.1145/192218.192221.

[20] B. P. Miller, L. Fredriksen and B. So, An empirical study of the reliability of unix utilities, *Commun. ACM* **33**, 12, pp. 32–44 (1990), doi:10.1145/96267.96279, http://doi.acm.org/10.1145/96267.96279.

[21] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo and T. Y. Chen, Automated functional testing of online search services, *Softw. Test. Verif. Reliab.* **22**, 4, pp. 221–243 (2012), doi:10.1002/stvr.437, http://dx.doi.org/10.1002/stvr.437.

[22] S. Hangal and M. S. Lam, Tracking down software bugs using automatic anomaly detection, in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02. ACM, New York, NY, USA, ISBN 1-58113-472-X, pp. 291–301 (2002), ISBN 1-58113-472-X, doi:10.1145/581339.581377, http://doi.acm.org/10.1145/581339.581377.

[23] V. Chandola, A. Banerjee and V. Kumar, Anomaly detection: A survey, *ACM Comput. Surv.* **41**, 3, pp. 15:1–15:58 (2009), doi:10.1145/1541880.1541882, http://doi.acm.org/10.1145/1541880.1541882.

[24] H. Jin, Y. Wang, N. W. Chen, Z. J. Gou and S. Wang, Artificial neural network for automatic test oracles generation, in *Computer Science and Software Engineering, 2008 International Conference on*, Vol. 2, pp. 727–730 (2008), doi:10.1109/CSSE.2008.774.

[25] K. K. Aggarwal, Y. Singh, A. Kaur and O. P. Sangwan, A neural net based approach to test oracle, *SIGSOFT Softw. Eng. Notes* **29**, 3, pp. 1–6 (2004), doi:10.1145/986710.986725, http://doi.acm.org/10.1145/986710.986725.

[26] W. K. Chan, M. Y. Cheng, S. C. Cheung and T. H. Tse, Automatic goal-oriented classification of failure behaviors for testing xml-based multimedia software applications: An experimental case study, *J. Syst. Softw.* **79**, 5, pp. 602–612 (2006), doi:10.1016/j.jss.2005.05.031, http://dx.doi.org/10.1016/j.jss.2005.05.031.

[27] M. Vanmali, M. Last and A. Kandel, Using a neural network in the software testing process, *International Journal of Intelligent Systems* **17**, 1, pp. 45–62 (2002), doi:10.1002/int.1002, http://dx.doi.org/10.1002/int.1002.

[28] S. Shahamiri, W. Wan-Kadir, S. Ibrahim and S. Hashim, Artificial neural networks as multi-networks automated test oracle, *Automated Software Engineering* **19**, 3, pp. 303–334 (2012), doi:10.1007/s10515-011-0094-z, http://dx.doi.org/10.1007/s10515-011-0094-z.

[29] F. Tsimpourlas, A. Rajan and M. Allamanis, Learning to encode and classify test executions, (2020), arXiv:2001.02444 [cs.SE].

[30] F. Wang, L. Yao and J. Wu, Intelligent test oracle construction for reactive systems without explicit specifications, in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pp. 89–96 (2011), doi:10.1109/DASC.2011.39.

[31] S. Parsa, S. Arabi Nare and M. Vahidi-Asl, *Early Bug Detection in Deployed Software Using Support Vector Machine*. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-89985-3, pp. 518–525 (2009), ISBN 978-3-540-89985-3, doi:10.1007/978-3-540-89985-3_64, http://dx.doi.org/10.1007/978-3-540-89985-3_64.

[32] S. Parsa and S. A. Naree, Software online bug detection: applying a new kernel method, *IET Software* **6**, 1, pp. 61–73 (2012), doi:10.1049/iet-sen.2010.0057.

[33] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche and R. Subramanyan, Automating image segmentation verification and validation by learning test oracles, *Information and Software Technology* **53**, 12, pp. 1337–1348 (2011), doi:http://dx.doi.org/10.1016/j.infsof.2011.06.009, http://www.sciencedirect.com/science/article/pii/S095058491100156X.

[34] Y. Brun and M. D. Ernst, Finding latent code errors via machine learning over program executions, in *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04. IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2163-0, pp. 480–490 (2004), ISBN 0-7695-2163-0, http://dl.acm.org/citation.cfm?id=998675.999452.

[35] M. Haran, A. Karr, A. Orso, A. Porter and A. Sanil, Applying classification techniques to remotely-collected program execution data, *SIGSOFT Softw. Eng. Notes* **30**, 5, pp. 146–155 (2005), doi:10.1145/1095430.1081732, http://doi.acm.org/10.1145/1095430.1081732.

[36] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil and S. Fouche, Techniques for classifying executions of deployed software to support software engineering tasks, *IEEE Transactions on Software Engineering* **33**, 5, pp. 287–304 (2007), doi:10.1109/TSE.2007.1004.

[37] D. Lo, H. Cheng, J. Han, S.-C. Khoo and C. Sun, Classification of software behaviors for failure detection: A discriminative pattern mining approach, in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09. ACM, New York, NY, USA, ISBN 978-1-60558-495-9, pp. 557–566 (2009), ISBN 978-1-60558-495-9, doi:10.1145/1557019.1557083, http://doi.acm.org/10.1145/1557019.1557083.

[38] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun and B. Wang, Automated support for classifying software failure reports, in *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03. IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-1877-X, pp. 465–475 (2003), ISBN 0-7695-1877-X, http://dl.acm.org/citation.cfm?id=776816.776872.

[39] P. Francis, D. Leon, M. Minch and A. Podgurski, Tree-based methods for classifying software failures, in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04. IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2215-7, pp. 451–462 (2004), ISBN 0-7695-2215-7, doi:10.1109/ISSRE.2004.43, http://dx.doi.org/10.1109/ISSRE.2004.43.

[40] J. F. Bowring, J. M. Rehg and M. J. Harrold, Active learning for automatic

classification of software behavior, in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04. ACM, New York, NY, USA, ISBN 1-58113-820-2, pp. 195–205 (2004), ISBN 1-58113-820-2, doi:10.1145/1007512.1007539, `http://doi.acm.org/10.1145/1007512.1007539`.

[41] C. Mao and Y. Lu, *Extracting the Representative Failure Executions via Clustering Analysis Based on Markov Profile Model*. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-31877-4, pp. 217–224 (2005), ISBN 978-3-540-31877-4, doi:10.1007/11527503_26, `http://dx.doi.org/10.1007/11527503_26`.

[42] G. K. Baah, A. Gray and M. J. Harrold, On-line anomaly detection of deployed software: A statistical machine learning approach, in *Proceedings of the 3rd International Workshop on Software Quality Assurance*, SOQUA '06. ACM, New York, NY, USA, ISBN 1-59593-584-3, pp. 70–77 (2006), ISBN 1-59593-584-3, doi:10.1145/1188895.1188911, `http://doi.acm.org/10.1145/1188895.1188911`.

[43] R. Almaghairbe and M. Roper, Automatically classifying test results by semi-supervised learning, in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 116–126 (2016), doi:10.1109/ISSRE.2016.22.

[44] M. Roper, Using machine learning to classify test outcomes, in *IEEE International Conference On Artificial Intelligence Testing, AITest 2019, Newark, CA, USA, April 4–9, 2019*. IEEE, pp. 99–100 (2019), doi:10.1109/AITest.2019.00009, `https://doi.org/10.1109/AITest.2019.00009`.

[45] W. Dickinson, D. Leon and A. Podgurski, Finding failures by cluster analysis of execution profiles, in *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pp. 339–348 (2001a), doi:10.1109/ICSE.2001.919107.

[46] W. Dickinson, D. Leon and A. Podgurski, Pursuing failure: The distribution of program failures in a profile space, in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9. ACM, New York, NY, USA, ISBN 1-58113-390-1, pp. 246–255 (2001b), ISBN 1-58113-390-1, doi:10.1145/503209.503243, `http://doi.acm.org/10.1145/503209.503243`.

[47] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff and C. Yang, Estimation of software reliability by stratified sampling, *ACM Trans. Softw. Eng. Methodol.* **8**, 3, pp. 263–283 (1999), doi:10.1145/310663.310667, `http://doi.acm.org/10.1145/310663.310667`.

[48] S. Yoo, M. Harman, P. Tonella and A. Susi, Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge, in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM Press, pp. 201–211 (2009).

[49] S. Yan, Z. Chen, Z. Zhao, C. Zhang and Y. Zhou, A dynamic test cluster sampling strategy by leveraging execution spectra information, in *Proceedings of the 2010 Third International Conference on Software Testing, Verifi-*

cation and Validation, ICST '10. IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3990-4, pp. 147–154 (2010), ISBN 978-0-7695-3990-4, doi:10.1109/ICST.2010.47, http://dx.doi.org/10.1109/ICST.2010.47.

[50] W. Masri, A. Podgurski and D. Leon, An empirical study of test case filtering techniques based on exercising information flows, *IEEE Transactions on Software Engineering* **33**, 7, pp. 454–477 (2007), doi:10.1109/TSE.2007.1020.

[51] R. Almaghairbe and M. Roper, Building test oracles by clustering failures, in *10th IEEE/ACM International Workshop on Automation of Software Test, AST 2015, Florence, Italy, May 23–24, 2015*, pp. 3–7 (2015), doi:10.1109/AST.2015.8, https://doi.org/10.1109/AST.2015.8.

[52] R. Almaghairbe and M. Roper, Separating passing and failing test executions by clustering anomalies, *Software Quality Journal* **25**, 3, pp. 803–840 (2017), doi:10.1007/s11219-016-9339-1, https://doi.org/10.1007/s11219-016-9339-1.

[53] R. Almaghairbe and M. Roper, An empirical comparison of two different strategies to automated fault detection: Machine learning versus dynamic analysis, in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 378–385 (2019), doi:10.1109/ISSREW.2019.00099.

[54] R. Almaghairbe, M. Roper and T. Almabruk, Machine learning techniques for automated software fault detection via dynamic execution data: Empirical evaluation study, in *Proceedings of the 6th International Conference on Engineering and MIS 2020*, ICEMIS'20. Association for Computing Machinery, New York, NY, USA, ISBN 9781450377362 (2020), ISBN 9781450377362, doi:10.1145/3410352.3410747, https://doi.org/10.1145/3410352.3410747.

[55] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, The daikon system for dynamic detection of likely invariants, *Sci. Comput. Program.* **69**, 1–3, pp. 35–45 (2007), doi:10.1016/j.scico.2007.01.015, http://dx.doi.org/10.1016/j.scico.2007.01.015.

https://doi.org/10.1142/9789811239922_0011

Chapter 11

Intelligent Risk Based Analysis Methodology

Eli Menasheof
Intel Corporation, Jerusalem, Israel
eli.menasheof@intel.com

11.1 Introduction

Traditionally, success in software project management has been associated with the ability to find balance among the project's cost (funding and resources), time (schedule and deadlines), and quality (scope and features), that constrain overall project delivery [12]. The main question is whether it is possible to achieve a successful project without trade-offs or sacrificing quality given these constraints.

Since 2000, this challenge has received considerable attention in both academia and industry. In the literature, in order to cope with this challenge, researchers have proposed software-testing-related methods such as **Risk-Based Testing (RBT)** and **Test Case Prioritization (TCP);** software development frameworks, like **Agile** and **Continuous Integration (CI)**; and scientists have suggested some **Artificial Intelligence (Al)** methods. These methods suggest various solutions, but the most common mindsets for risk analysis fundamentals in software testing are "prevention is always better than cure," as recommended by [30], and also "no pain, no gain." This means, with more investment prior to the project on collecting available data, analyzing them, and consulting with experts, one will achieve a better long-term risk strategy plan, which will justify the investment. In practice, this kind of solution usually leads to a low return on investment (ROI) because the pain, which is in this context the

investment, comes ahead of achieving the potential gain, i.e., the return. It might also lead to negative ROI that reflects in the tendency of these methods to avoid changes during the project in order to reduce continuous maintenance efforts by key experts. This informed mindset has a negative impact on project success, and on customer satisfaction because customers usually ask for further changes during the project. As a result, instead of focusing on the challenge, which is project success and customer satisfaction, the major investment by these methods is on a new preparatory step they added that wasn't part of the original project scope, and is now a prerequisite for the project. In addition, existing RBT methods mostly prioritize testing a single application-under-test and cannot cope with the interoperability of a large number of applications, as the cost would be even more expensive. Is it possible to reduce the huge scope without compromising functional coverage, and at the same time expose the most high-risk areas with its defects?

This chapter proposes **i**ntelligent **R**isk **B**ased **A**nalysis methodology **(iRBA)**, challenging this status quo. This model changes the rules of the game by suggesting an original approach to address the challenge. It is the first time that a method suggests "gain without pain," or at least with minimum pain, by intelligent effective utilization of it. iRBA suggests a practical AI-powered data analysis learning method, built up in real time according to the behavior and environment of the project. Any significant data, event, change, or risk that can impact the project's priority are iRBA's bread and butter toward achieving immediate gain. Continuously, iRBA measures and adapts to reality and always maintains the best possible strategy plan, progress, coverage, and quality.

11.2 Motivation and Related Works

Risk-Based Testing. To cope with this challenge, researchers proposed several RBT methods. A good analogy by [14] compares RBT to eating food. Most of the existing RBT methods are based on Bach's attitude. With more investment on collecting data (*about our health*), analyzing them, and consulting with experts (*nutritionists, etc.*), one will achieve a better long-term risk strategy plan (*diet menu*) that will justify the investment (*ensure our health durably).* In the ideal world this sounds like a great solution. But, even if you succeed in creating the best plan (*diet menu*), it is not as effective as it could be. The project (*our life*) is dynamic and much more complex. Many events can impact this plan. The

requirements may be changed many times by customers, and the schedule and resources may be changed during the project (*a lot of things impact on your health like daily physical fitness, exposure to others, or contagious disease, health changes, etc.*). It's also impractical to do dynamically, as it requires too much investment to run it frequently. We can't ask most of our key experts to be dedicated to updating the plan all the time.

On the one hand, RBT has the potential to guide the project by improving the software test process, as it helps to optimize the allocation of resources and provides decision support for management based on the plan. Once a plan is made before the project, all the teams reach an agreement and follow it, taking the risk-factor into account. On the other hand, RBT gives only a partial solution to the challenge, and the return on investment is low. The overall effort can be quite high, especially when key project members are involved. The process requires gleaning all the data after all members have submitted their opinion. Their limited availability may impose serious restrictions concerning possible risk estimation and consolidation methods.

In addition, the plan and its conclusions are based only on data available before the project, which means it's mostly based on estimates, as some data are still not clear at this point, and they cannot predict the future. In this context, [14] reminds that all you really have at the beginning of a project are rumors of risks, and the RBT is going to be incomplete and inaccurate, and may really be very wrong during the project. It becomes even more challenging when such estimates need to be conducted frequently throughout the lifecycle of a project. Although there are ample advantages to covering all eventualities, there is also a considerable price to pay: it makes both the plan-creation process and the decision-making process slower and more cumbersome. As a mitigation, in order to reduce continuous maintenance efforts by key experts, the mindset of these existing RBT methods is risk averse and change avoidance during the project. That means, once a plan is made and the project has started, accept minimum new requirements or changes that were not taken into account in the initial plan. This informed mindset has a negative impact on the satisfaction of customers, who usually ask for new requirements and changes during the course of a project.

Agile [20] claimed to be a framework that provides flexible responsiveness to changes when using the RBT methods. In Agile there is a fixed schedule and fixed resources, thus, when things don't work according to the plan, the scope needs to be reduced. In this case, the scope should

include the highest priority items in the product backlog for maximizing the value yielded by the project. In practice, RBT in Agile can be challenging, too, when scaling projects; it also requires constant refactoring as changes are frequent and even more so if the scope is not properly defined from the early stages of the project. There is uncertainty in prioritization of test cases, too much time spent clarifying and communicating test requirements, a huge backlog of test cases, and rising test environment costs.

Continuous Integration is another cost-effective software development framework commonly used in the industry [19, 21], where developers frequently integrate code changes in the main branch from which new software releases are deployed directly into production [26]. Testing of new software release candidates in CI is a crucial activity to detect defects as early as possible and to ensure that changes do not adversely affect existing functionality. However, executing an entire regression test suite after every code change is often costly in large software projects.

Test Case Prioritization. To deal with this challenge, researchers have suggested an additional level of optimization: TCP methods [34], to order the test cases [8] and select the top-k ranked test cases as a subset of tests suite that must be run to lower costs without overly reducing effectiveness [13]. The most recent systematic literature review that classified TCP methods was performed by [15]. They concluded that each existing method has specific potential limitations and can be improved both in terms of data used and execution process. The challenge of applying most of them in practice lies in their complexity and the computational overhead typically required to collect and analyze different test case parameters needed for prioritization. In addition, these methods often either underestimate or overestimate the set of test cases needed [13].

Artificial intelligence. To cope with this challenge, scientists have suggested using AI. Using AI methods improves the defect prediction and TCP, but there is a steep price to pay, as key experts are required to provide quality data and invest resources to collect history data, collecting as input to be used as training data, in order to gain experience from previous data.

[2] suggest using Machine Learning (ML) and multiple heuristic techniques, while the requisite input is the code coverage information, text path similarity, text content similarity, failure history, and test case age. [4] suggest using Artificial Neural Networks (ANN), while the essential input is test case metrics like length, command frequency, and parameter use frequency. [22] propose case-based reasoning using ML, while the required input is the prioritization indexes, priority information from the user,

user-defined pairwise priority relations on test cases, and structural coverage information. [11] recommend using Reinforcement Learning (RL), while the mandatory input is information on test cases, including the test case duration, the last execution, and a failure history. [23] advise applying Support Vector Machine (SVM), and [10, 35] suggest using Natural Language Processing (NLP), while the required input is the history data. [17] advocate using ML, while the necessary input is all previous defects metrics, as well as newly defined metrics by the method. [29] suggest using Deep Neural Networks, while the input required is a list of correct and incorrect spreadsheets and previous defects repositories. To group test cases into clusters to be used by defect prediction and TCP, [1] suggest using ML methods, while the required input is structural coverage information, number of mutants killed, and mutation score associated with each mutation operator. [25] suggest using semi-supervised clustering, while the needed input is data labels or data constraints, to be provided by key experts.

Dozens of ML-based studies to improve defect prediction and TCP were reviewed by [5,32]. They concluded there is much left to improve ML-based methods quality to improve defect prediction and TCP. They indicate the main common drawback of these methods is their dependency on a large amount of data as required input for the training. The success of all these methods depends on the quality of the data, which means definition of the metrics by key experts. Another drawback is their inability to discover new types of defects, such as defects that were not defined in advance nor as part of the training data.

Most of these methods apply systematic prioritization algorithms and improve the defect prediction by learning on the entire training data set at once, which means they are not well adapted to changes. The learning process and effectiveness using the reward functions is not optimal, as the agent interacts in a dynamic execution environment, and gets rewards with dynamic impact. That means, the same action for the same test case does not always yield the same reward. The importance of some test cases might be changed over others during the project. New test cases are added to cover new requirement changes, while other test cases might be removed. Some test cases are crucial in certain time periods, as they cover areas mostly important to the customers but become less important with low priority, following unplanned focus change during the project. Test cases are selected by these methods to expose defects, based on previous experiences because they are likely to fail, but no longer fail when this area becomes irrelevant as it grew more stable with previous code fixes. Test cases that were not

selected for execution and removed from the scope, and their pass would be expected, based on previous experiences, because they are likely to pass, no longer pass, following unplanned new requirement change requests by the customer that also impacted the area to which this test case belongs.

Another limitation is the required history collecting as the input for the process. Theoretically, long history provides more data and allows better knowledge about the risky areas and defect distribution. Furthermore, it has negative impact on the performance time, as it requires processing of much more data.

[11] raise the question of how long the history of test case results should be for a reliable indication. He concluded a longer history can reduce performance due to a higher complexity, and as it requires more data to adapt. The agent has to learn to handle earlier execution results differently from more recent ones, which is the most relevant result, for example by weighting or aggregating them. These limitations make it harder for the learning agent to discover a feasible prioritization strategy.

Interoperability. Matters get even worse in the case of complex environments with multiple applications-under-test. Existing RBT methods prioritize mostly testing of a single application-under-test and cannot cope with the interoperability of a large number of applications. [3, 33, 37], require a risk assessment phase by experts. It is too costly and impractical manually to assess the criticality of possible defects in this complex environment. [18, 28] propose interoperability test cases generation methods, and reduce the scope without compromising functional coverage, but can't expose the most high-risk areas. In addition, the number of generated test cases depends on the global model's complexity.

11.3 iRBA Solution

iRBA suggests a novel data analysis learning method, built up in real time according to the behavior and environment of the project. iRBA is based on unique techniques and elements, combined with elements of RL Q-learning and online machine learning. RL is well-tuned to design an adaptive method capable of learning from its experience of the execution environment. A more in-depth introduction can be found in [24].

iRBA method uses a unique prioritization technique combined with online machine learning in which data becomes available in a sequential order and is used to update the best predictor at each step, constantly learning during its runtime. This is also appropriate where indicators for failing

test cases can change over time due to unplanned changes in the dynamic environment. iRBA method can progressively improve its efficiency from observations of the effects of its actions. iRBA continuously measures and adapts to reality and achieves the best predictability possible.

11.3.1 *iRBA Unique Techniques and Elements*

The following list of iRBA techniques reflects those practices and their interpretation most commonly encountered by intelligent methodology.

(1) **irba** represents the probability of the test case to find defects.
(2) **iRBA-Link** or **x** represents a common linkage criteria item that enables grouping test cases that are impacted by this item.
(3) **Impacted(x)** represents a function that returns all test cases that are impacted by the linkage criteria x.
(4) **iRBA-Reward-Functions** define the functions and the impact by the positive or negative reward that triggered by list of significant events that might impact on test cases priority, which are part of the daily user's activities during the project.
(5) **iRBA-Reward** represents any new data provided to the agent immediately every time a new significant event happens, for adaptation and agent's policy improvement.
(6) **iRBA-Prioritization** continuously re-prioritizes the impacted test cases from the set of all test cases immediately, with any new significant event, using iRBA-Reward-Functions. It means, the next test case to be executed might be changed as it's defined in real time.
(7) **iRBA-Queue-Management** or **iQ()** schedules the individual test case with the highest priority in the sequence, at this moment, to be executed next. As a result, iRBA succeeds in overcoming outliers, as they will most likely be caught by the impact of other relevant events, as this kind of test will never be the one with the highest priority.
(8) **iRBA-Dynamic-Scope** keeps the set of all the test cases in the scope during the whole project. Instead of selecting what test cases to execute or cut in advance, iRBA enables any test case from the bottom part of the list gradually to move up to the top part, and vice versa, according to the changing environment that reflects their reliable and actual priority at this moment. That way, iRBA focuses on the highest priority test case at any moment, and always maintaining the best possible coverage and quality, according to the given schedule and resources. Additionally, it enables effective decision-making in real

time regarding scope definition and reduction and resource allocation, due to unplanned changes.

(9) **iRBA-3D-Graph** represents a very powerful database and capability that enables grouping test cases from multiple applications-under-test into mutual prioritized clusters, and detects major interoperability defects much earlier.

(10) **Non-iRBA** represents the other existing traditional and modern methods that are reviewed in Sec. 11.2

11.3.2 *Test Case Prioritization — Problem Formulation, Statement and Definition*

This section introduces the notations proposed by [9] that are used in the rest of the paper and present the addressed challenge in a formal way.

Let T be an unprioritized set of all the test cases developed for validating the product. t_i is a test case in the set $t_i \in T : \{t_1, t_2, ..., t_n\}$.

Let PT be the set of all possible permutations of T, and f is a function that calculates the probability of finding defects sooner. $f(T')$ is maximized when T' is an optimal prioritized sequence of test cases that finds defects as early as possible.

Problem:

Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f'(T') \geq f'(T'')]$

With non-iRBA methods, TCP target is finding T', such that $f(T')$ is maximized prior to the test cycle. With iRBA method, iRBA-Prioritization target is keeping T' prioritized at any moment during the whole project, so $f(T')$ is always maximized.

11.3.3 *TCP Method Using RL — iRBA Vs. Non-iRBA*

Figures 11.1 and 11.2 show how the iRBA method and non-iRBA methods solve the prioritization problem using RL. *State* represents a test case t_i with its parameters. *Action* returns the recommended sub test suite called T'_c to be run first in test cycle c in order to expose early defects. This recommendation is based on an optimization process that was performed by the agent. $T'_c \subseteq T \quad c = 1...m$.

With non-iRBA methods, the decision about the scope coverage comes prior to the test cycle, and is based on a one time analysis of results from the previous run. This means, there is an optimization process by training the algorithm, or by TCP, to order and select only the high priority test cases that would fail with high accuracy as the new sub test suite T'_c

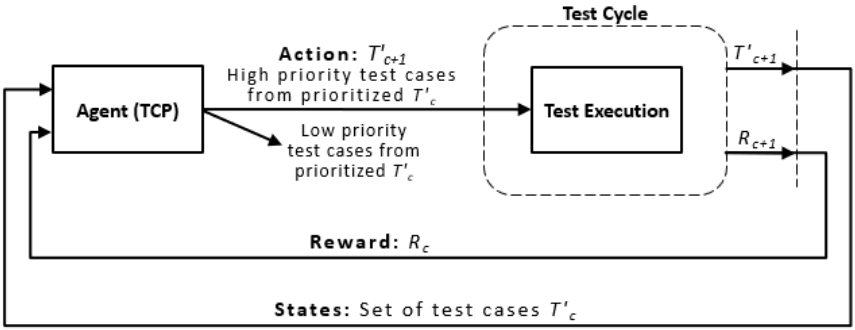


Fig. 11.1 Non-iRBA TCP using RL.

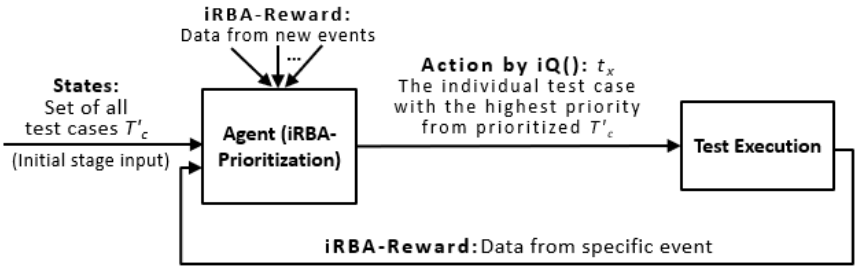


Fig. 11.2 iRBA-Prioritization using RL.

depends on the cycle timeframe that was defined in advanced plus other project limitations. Only this subset is the input for the test execution, which means, the number of test cases is decreased in every test cycle. $T \supseteq T'_1 \supseteq T'_2 \supseteq \ \ldots \ \supseteq T'_m$.

Another disadvantage is the optimization level of the testing coverage during the cycle. If the defined test suite is completed before the pre-set time, the time left is wasted in vain, as no more test cases would be added at this point. On the other hand, if the defined test suite was not completed on time, due to unplanned blockers, the test cases that covered are not necessarily the most important at this stage, but those that were the most important at the analysis time prior to the cycle.

With iRBA method, the decision about the scope, coverage, and the optimization process, is based on real time analysis and is much more effective. This is achieved by keeping the set of all the test cases prioritized at any moment in all the executions $T'_1 = T'_2 = \ \ldots \ = T'_m$ and by using the **iRBA-Queue-Management** during the test execution.

iRBA-Dynamic-Scope also enables exposing critical defects that are well-hidden behind test cases that were initially defined as low priority. As shown in the industrial case study in Sec. 11.5, these kinds of test cases indeed exposed many critical defects much earlier.

With non-iRBA methods, the reward is calculated and fed back to the agent once having completed running the test cycle. Let R_c be non-iRBA reward that represents the results data of all the test cases from the test cycle c. These results are only for the selected subset T'_c which means, the total number of rewards is dependent on the total number of the test cases in the test suite, which is decreased in every test cycle $R_1 \supseteq R_2 \supseteq \ ... \ \supseteq R_m$.

In some non-iRBA methods, the length of the history results data is longer, which means results are kept from many previous test cycles or even previous projects for future analysis.

With iRBA method, the iRBA-Reward is provided to the agent immediately with any new meaningful event, to adapt its experience and policy for future actions. The total number of rewards is not related to the number of test cases but to the events.

With non-iRBA methods, the result of test cases from previous execution, or from long history data, impact only on the priority of this specific test. In the case of positive reward, it gets higher priority to be included in the next Test Schedule, while in the case of negative reward it gets lower priority.

With iRBA method, the impact on the priority is on all the test cases that are impacted by the latest change, based on iRBA-Reward-Functions definitions.

Let *irba* be a number that represents the probability of a test case to find defects, where a greater value shows a greater defect detection rate. Each test case has an iRBA value $t_i.irba$.

The initial *irba* score for all the test cases is 1, and this score increased in the initial step, based on the test case's importance level, critical, high, medium, and low.

$$\forall t_i \in T \ : \ t_i.irba = 1 + t_i.importance_{level}$$

Let $link[]$ be an array of all the linkage criteria items that are defined for the project. $t_i.link[x]$ is either true when the test case t_i is impacted by linkage criteria x, or $false$ if not. Let $Impacted(x)$ be a function that returns all the test cases that are impacted by the linkage criteria x. In this context, it could be, all test cases that use the same mutual resource depend on a specific external component, or validate the same sub-feature or code area.

$$\forall t_i \in T \qquad Impacted(x) = \begin{cases} t_i.link[x] = true, & t_i \in Impacted(x) \\ t_i.link[x] = false, & t_i \notin Impacted(x) \end{cases}$$

11.3.4 *The iRBA-Reward-Functions*

Generally, the reward functions should lead to early defect detection to steer the agent's behavior [36]. The reward should impact on the prioritization of the test cases and the execution order. Every passed test case reduces its quality if it precedes a failing test case. Failing test cases should be executed first and as early as possible. In order to achieve this target and focus on areas most likely to have defects, iRBA's reward functions are using existing software testing principles [6] that offer general guidelines common for all testing.

Each reward function represents significant events that occur many times during the project, and should contribute to the project's success by impacting immediately upon prioritization. Each reward function has its weight, which represents the positive or negative reward that was defined and might be changed by the RL algorithm, according to learning over time and policy improvement.

Let w_e be the weight of the positive or negative reward of event e on the impacted test cases. It means that it defines how much to increase or decrease their *irba* score. In Sec. 11.5, there is an example table that represents the optimal weight that is defined and used as part of the industrial case study.

11.3.4.1 *iRBA Reward Function — Test Case Passed Event*

The guiding principle for this factor is the 'pesticide paradox' [6] — i.e., if the same test cases are running over and over again, eventually the same test case will no longer find any new defects. It is recommended to review and revise this test case or invest the limited resources to focus on other test cases in riskier areas first, to find new defects.

In the event a test case passed without finding defect, it has negative impact on the target, so the iRBA value of the test is decreased by the negative reward. The impact is only on the test case itself. There is no impact on other test cases because the passing of a specific test case cannot bear witness to other test cases. To improve the accurate estimate, each test case has $t_i.running$, which represents the running number of this test case during the project. This means, the impact level is dependent on the running number in order to overcome outliers. For example, there should

be a different impact among test cases that succeed in being the one with highest priority many times during the project, compared with test cases that run only once based on its importance probability.

$$t_i.irba = t_i.irba \cdot \left(1 - \frac{w_e}{t_i.running}\right)$$

11.3.4.2 *iRBA-Reward-Functions — New Defect Event*

The guiding principle for this factor is 'defect clustering' [6] — i.e., the tendency of defects to cluster together and that the greater number of defects found in any given feature indicates a greater number of defects is expected to follow. The rationale for the defects on a specific area can be a complex algorithm, bad specifications, an inexperienced developer, or complex interactions with other parts of the product.

In the event the test case failed and exposed new defects with new linkage criteria x, it has positive impact on the target, so the iRBA value of all the test cases that are impacted by x is reinforced by positive reward.

Additionally, to improve the accurate estimate, the reinforcement of iRBA values is relative to the severity of the defect, correspondingly.

$$\forall t_i \in Impacted(x) \ : t_i.irba = t_i.irba \cdot w_e$$

11.3.4.3 *iRBA-Reward-Functions — Defect Rejected Event*

In case the defect is rejected as not a real defect that was submitted by mistake, the method is to undo the previous impact of this defect and return to its original iRBA value.

$$\forall t_i \in Impacted(x) \ : t_i.irba = \frac{t_i.irba}{w_e}$$

11.3.4.4 *iRBA-Reward-Functions — New Code Integration Event*

Every time a developer integrates a new change to the code, by fixing a defect or implementing a requirement, there is a new linkage criteria x as part of the code integration process, that reflects the most impacted areas to focus on based on his insights from investigating this specific change.

It has positive impact on the target because it indicates the newly released version that contains the new code that had never been tested and needs more attention.

In this case, the iRBA value of all the test cases that are impacted by x is reinforced, by positive reward.

Additionally, to improve the accurate estimate, the reinforcement of iRBA values is relative to the severity of the defect, correspondingly.

$$\forall t_i \in Impacted(x) \ : t_i.irba = t_i.irba \cdot w_e$$

11.3.4.5 *iRBA-Reward-Functions — New Test Case Creation Event*

In the event a new test case is created during the project and added to the scope, it is indicated on the new feature, or new flow, or area in the code that was not yet tested, and has positive impact on the target. In this instance, the iRBA value of the test case itself is reinforced after the initial step.

$$t_i.irba = (1 + t_i.importance_{level}) \cdot w_e$$

11.3.4.6 *iRBA-Reward-Functions — Customer Feedback or Unplanned Change Event*

This function enables effective use of customer feedback during the project to drive product quality. It could be as a result of a new requirement change request from the customer, new feature or a decision on focus change.

In case the request is to focus more on a specific area, there is a new linkage criteria x, and it has positive impact on the target because it indicates a new area that never has been tested, or existing areas that need more attention. In this situation, the iRBA value of all the test cases that are impacted by x is reinforced.

$$\forall t_i \in Impacted\,(x) \ : t_i.irba = t_i.irba \cdot w_e$$

In case the request is to focus less on a specific area x, it is recommended to invest the limited resources to focus on other areas that are important to the customer first. In this circumstance, the iRBA value of all the test cases impacted by x is decreased by the negative reward.

$$\forall t_i \in Impacted\,(x) \ : t_i.irba = \frac{t_i.irba}{w_e}$$

11.4 iRBA Benefits Demonstrations

Introducing state-of-the-art iRBA methodology delivers multiple benefits. It enhances testing quality, team efficiency and agility as well as being far more predictable in achieving milestones. This intelligent method enables

achieving a successful project without sacrificing quality, given the constraints among the project's cost, time, and quality. Only a selected few of iRBA advantages are presented here. Those presented are the most "visibly measurable" and easy to verify by outsiders.

11.4.1 *Project's Cost Challenge*

11.4.1.1 *Become Much More Effective with Cost-Effective Manpower and Limited Resources*

The recommended mindset behind iRBA method is maximizing gain and minimizing pain, while keeping the process and interfaces, managing as lean and agile as possible, only with those processes that achieve gain immediately for project success and quality.

As concluded by [7], the major pain of the RBT methods are the prerequisite steps, experts' availability, integration into an established test process, and reliability in complex systems.

This leads iRBA to cut back on unnecessary expenses and save

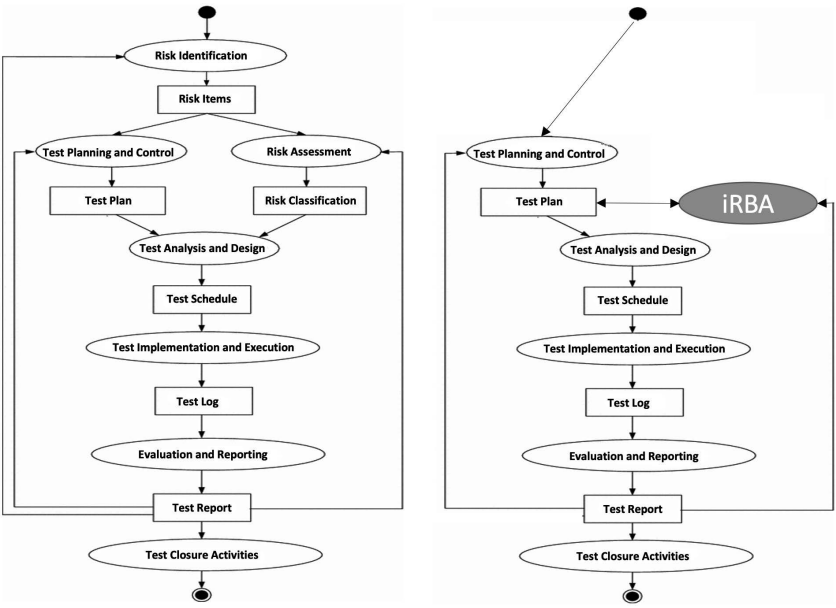


Fig. 11.3 Generic RBT process. Figure is taken from [7]. Some changes made to show the generic iRBA process.

meaningful time and resources, by reducing the pain from the process without having a negative impact on the end result, as shown in Fig. 11.3.

The cost reduction includes freeing the domain experts and senior engineers by removing the RBT prerequisites phases from the scope, including Risk Identification and Risk Assessment by experts, mentioned by [31] as critical cost factors. Another pain is the Test Schedule process and its preparation steps by existing RBT or TCP methods before each Execution Cycle. iRBA suggests a much more effective and lean process, by handling both RBT and TCP challenges at the same time, without performing computationally intensive operations during prioritization. In iRBA, data collection and analysis are not separate stages of the project. They are embedded in the project process and happen all the time during the entire Software Development Life Cycle (SDLC). It enables reflecting feedback by customers and the insights from testers, developers, and architects, who are valuable resources for the software project's success and quality. There are special time points in their daily work in which they have unique insights with high impact on the project success and quality. Based on the iRBA mindset, these time points are the optimal times to collect the required data for the method, as the effort required in this case is negligible. For example, one of the required data to collect by iRBA and use as part of the rewards functions, is linkage criteria, which is a common item, enabling grouping impacted test cases. In this context, it could be all the test cases using the same mutual resource depend on specific external components, or validate the same sub-feature or code area. A developer who just integrated a new change to the code, by fixing a defect or implementing requirement, also fills as part of the code integration process, the linkage criteria, which reflects the most impacted areas by this change on which to focus the testing, based on his insights from investigation of this specific change. Linkage criteria could also be filled as part of the defect submission process by the tester, who just submitted a defect based on his insights from the initial debugging of the issue. A tester, who just created a test case as part of the test creation process, fills the importance level of the test case and linkage criteria, based on his insights from the requirement review with the architect of this specific test. Another limitation in most of the existing RBT methods, is the integration into an established test process, and the fact that a method suitable to one project may well be ineffective for another project. iRBA solution is agnostic framework applicable to any project, environment and SDLC model. iRBA built up in real time according to the behavior and environment of the project itself.

11.4.2 *Project's Quality Challenge*

11.4.2.1 *Effective Use of Customer Feedback to Drive Product Quality*

An indication for project success is customer satisfaction. When using iRBA, requirements change requests by customers during the project are more than welcome. Another meaningful impact on product quality and customer satisfaction can be attained by reducing escaped defects. iRBA framework enables utilizing customer pre-release or post-release feedback in real time and increases the priority in the impacted areas. That way, there is full partnership, enabling effective use of customer feedback and much faster response to any request from them.

11.4.2.2 *Risk Prioritization of Multiple Applications and Early Detection of Major Interoperability Issues*

iRBA also provides effective solutions for the complex integration and interoperability testing challenge. It enables prioritizing testing of multiple applications-under-test, reducing the number of test cases without compromising functional coverage, while at the same time exposing the riskiest areas.

In this context, the principle guideline is to use the 'divide and conquer' concept. The iRBA profit grows exponentially when it is used by a large number of teams. Each team can use iRBA separately for their applications-under-test testing. It enables them to focus on the riskiest issues at any moment and find the major defects sooner. Backstage, the **iRBA-3D-Graph** is created automatically.

Let d_j be one of the dimensions in the graph that represents applications-under-test j, including its set of all the test cases prioritized. It means, each d_j is a single applications-under-test which belongs to the multiple applications-under-test.

As shown before, iRBA enables using linkage criteria item x to group test cases from the same applications-under-test into clusters. In this context, $d_j.Impacted(x)$ function returns all the test cases that are impacted by the linkage criteria x, in applications-under-test d_j.

iRBA-3D-Graph provides a very powerful database and capability. It enables using the same linkage criteria item x that are in common to all the different applications-under-test, to group test cases from the different

applications-under-test into mutual clusters.

$$\bigcup_{j=1}^{m}(d_j.Impacted(x))$$

It returns all the test cases that are impacted by the same linkage criteria x, from all the applications-under-test. In this context, **iRBA-Queue-Management** schedules the individual test case with the highest priority in the sequence from each applications-under-test d_j at this moment of the project.

$$\bigcup_{j=1}^{m}(iQ(d_j.Impacted(x))$$

It returns j individual test cases with the highest priority from each applications-under-test d_j at this moment during the project that have a common linkage criteria x.

Quality interoperability and integration test case by combining two or more of these j test cases can be achieved this way. The generated test case is the peak quality test case with the highest probability of exposing defects related to the common linkage criteria x.

Based on this continuous output by **iRBA-Queue-Management**, a desired number of test cases is generated, while the scope is always updated according to the given schedule and resources. This is the key to successful projects, and the reason these test cases expose the critical and high defects much earlier.

As shown in the industrial case study in Sec. 11.5, risk prioritization of multiple applications and early detection of major interoperability issues are indeed feasible by iRBA.

11.4.2.3 *Quality Evaluation System*

Another important challenge for tracking project success is providing reliable status and progress that reflect quality. In a constantly changing environment, the status reports and graphs that generated based on non-iRBA methods are not effective enough and don't provide indications for real quality. These had negative impact on decisions during the project. iRBA centralizes and analyzes only the significant data and continuously measures and adapts to reality and makes adjustments in real time. In this way, it provides a reliable evaluation mechanism with quality indication.

11.4.3 *Project's Time Challenge*

Another indication for project success in the competitive equation is meeting customer needs on time. Time to market (TTM) improvement as measured by early defects detection and by delivering on schedule is meaningful toward achieving competitive advantage in the marketplace. As shown in the industrial case study in Sec. 11.5, there was meaningful TTM improvement across generations when using iRBA compared with non-iRBA.

11.5 Industrial Project Case Study

To realize the benefits of iRBA in a real-world environment, the research and results are from industrial projects used as a study object for analyzing and discussing the challenge. As in many other industrial projects, the SDLC is under time and budget constraints, that require continuous improvement of the established processes and methods. To support decisions about the focus and testing coverage and priority, the risk analysis activity model by [30] applied and the most popular non-iRBA methods implemented that have been upgraded over the years to create the optimal method. It also included all the non-iRBA prior requirements, history data from previous project generations analysis and dedicated key experts for the intensive preparation stages as well as during the project. For the sake of simplicity, let **Optimal-non-iRBA** be the optimal method that was used in the industry project.

In order to evaluate, a simulator was created to measure the quality between Optimal-non-iRBA vs. iRBA. For the experimental evaluation, the same set of parameters is used in all experiments, if not stated otherwise. These parameters are based on values from literature and experimental exploration. The comparison criteria include product quality (as measured by both the APFD quality indicator, and the defects count with their severity), cost reduction (as measured by freeing headcounts) and TTM across generations (as measured by early defects detection).

Average Percentage of Faults Detected (APFD) was introduced as a common quality indicator in [8] to measure the effectiveness of the methods and how rapidly a prioritized test suite detects defects. This evaluation metrics measures the quality via the ranks of defect detecting test cases in the test execution order. The value of APFD result can be from 0 to 1 where a bigger value shows greater fault detection rate. The equation for

calculating the APFD value is shown as follows:

$$APFD = 1 - \frac{TF1 + TF2 + ... + TFm}{nm} + \frac{1}{2n}$$

Where n represents the number of test cases, m represents the number of defects that were found, and TF_i is the earliest test case in the sequence which finds defect i.

The industrial data sets consist of historical data about test executions and their results for all the test cycles in the project. This complex and dynamic project included 36 components of embedded system technology, with 9,780 test cases. The experimental results have been computed on industrial data gathered over one year of project.

Table 11.1 presents the APFD results in two major test cycles, while the results were consistent throughout all the test cycles during the project. The results indicate better performance by iRBA method comparing to Optimal-non-iRBA, regarding prioritizing test cases and achieving higher APFD values.

Table 11.2 presents the optimal weight w_e that defined the positive or negative reward that used by iRBA-Reward-Functions as part of the

Table 11.1 APFD Results.

	Test Cycle 1	Test Cycle 2
Number of test cases (n)	9780	9780
Number of defects (m)	230	136
$TF1 + ... + TFm$ non-iRBA methods	241722	139464
$TF1 + ... + TFm$ iRBA methods	93834	73125
APFD non-iRBA	0.892	0.895
APFD iRBA methods	**0.958**	**0.945**

Table 11.2 iRBA-Reward-Functions Optimal Weights.

iRBA-Reward-Functions events	w_e
Test Case Passed	0.5
New Defect (critical, high, medium, low)	1.3, 1.2, 1.1, 1
Defect Rejected (critical, high, medium, low)	1.3, 1.2, 1.1, 1
New Code Integration	2.1
New Test Case Creation	2
Customer Feedback or Unplanned Change (positive, negative)	2,2
iRBA-Reward-Functions initial step	$importance_{level}$
Test case importance level (critical, high, medium, low)	5, 4, 2, 1

industrial case study. One of the limitations of APFD is the fact that it assumes defects were with equal severities, and test cases were with equal costs. As a result, another quality indicator function was created to measure effectiveness of the prioritization methods. To visualize and demonstrate the experiences and results from the industrial case study, using the new quality indicator function, a line graph is used, as shown in Fig. 11.4 and Fig. 11.5. This new function also takes additional conditions into account. In this case, the defects count is defined by the number of defects with respect to their severity: low, medium, high, and critical. The filled area in the graph represents the actual execution pace- it also takes into account special times like weekends, setup issues, and platforms upgrades and recovery. The non-iRBA Defect Score line represents what was really happened with Optimal-non-iRBA. The iRBA Defect Score line represents what would happen if test cases were run in iRBA order. Each graph represents the status of one test cycle during a real project. The results were consistent. In all test cycles, iRBA managed to find all the major defects much earlier.

Figure 11.4 shows that with iRBA all the major issues, critical and high priority defects, found much earlier, that is, within the first 44 days. In order to expose the same number of defects with non-iRBA methods, the

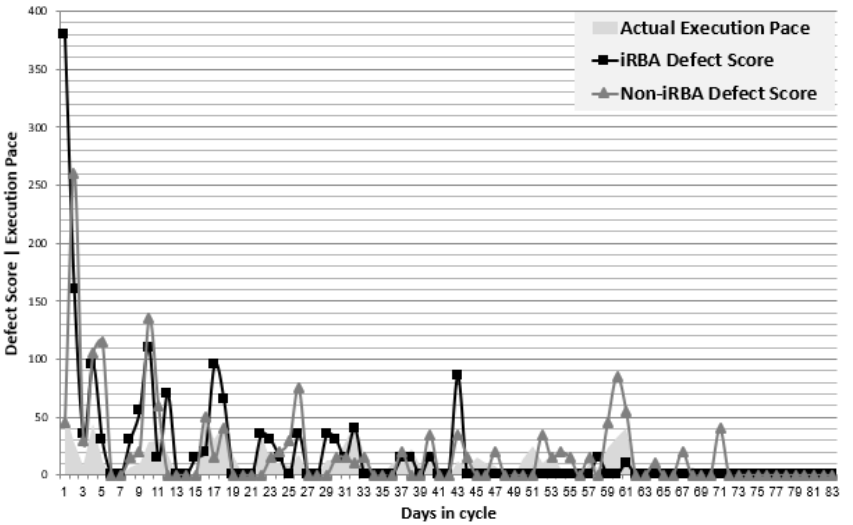


Fig. 11.4 Test Cycle 1: iRBA vs. non-iRBA comparison results.

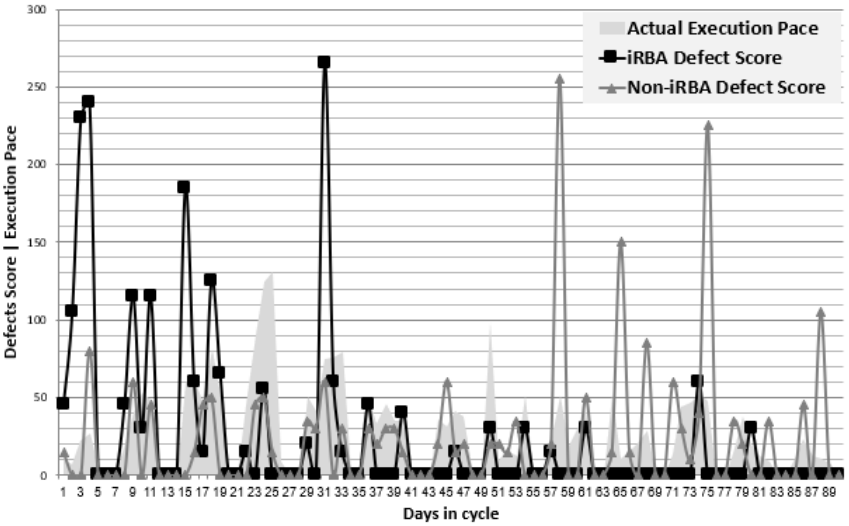


Fig. 11.5 Test Cycle 2: iRBA vs. non-iRBA comparison results.

test cycle should continue until the 71th day. Note that additional defects were also found using iRBA on the 61st day, however those defects had low exposure.

Figure 11.5 shows the same behavior. With iRBA, most of the major issues were found much earlier, before the 33rd day, compared to Optimal-non-iRBA, that required to continue until 89th day. TTM improvement by more than 30% across generations was measured by early defects detection. The major defects found by non-iRBA methods were also found by iRBA and much earlier in the timeline. The product quality improvement by more than 2X was measured by defects count. It is reflected by many critical and high well-hidden defects that were exposed much earlier by iRBA. Resource saving of 28% as measured by freeing headcounts. iRBA enabled the managers make up to date decisions regarding the schedule and resource allocation, by moving resources from low risk to high risk features and headcount reduction. It reduced validation effort and improved project predictability, cost, quality and efficiency. Another part of the case study object was to compare risk prioritization of multiple applications and early detection of major interoperability issues between Optimal-non-iRBA and iRBA.

Optimal-non-iRBA were applied by most experienced senior engineers,

who collected and analyzed huge amounts of data related to 6 applications under test, that their testing teams distributed in different geographical areas in the world, using top-of-the-line combinatorial testing tools.[1] iRBA used only its unique **iRBA-3D-Graph**. Both methods significantly reduced the number of test cases without compromising functional coverage, but Optimal-non-iRBA exposed very low numbers of critical and high defects compared to iRBA. The product quality improvement by more than 3X was measured by defects count. It's reflected by many critical and high defects that were exposed much earlier by iRBA. A resource saving of 60% as measured by freeing headcounts.

Table 11.3 Case Study Results: iRBA Improvements Based on the Comparison Criteria.

	iRBA improvements		
	Prioritization	**Interoperability**	**Measured by**
Quality improvement	2X	3X	Defects count; APFD
Resource saving	28%	60%	Freeing headcounts
TTM improvement	30%	25%	Early defects detection

11.6 Summary and Conclusions

A main outcome of the study was that iRBA methodology enables achieving a successful project without sacrificing quality given the constraints among the project's cost (funding and resources), time (schedule and deadlines) and quality (scope and features).

Continuously, iRBA was compared with traditional and modern software testing related methods and software development frameworks, including AI-based solutions, to cope with more and more existing software testing challenges. An important aspect in this context is that many of the proposed iRBA solutions have been tested as case studies in complex and dynamic industrial project environments in real-world.

The comparison criteria include product quality (as measured by both APFD quality indicator, and defects count and their severity), cost reduction (as measured by freeing headcounts) and TTM across generations (as measured by early defects detection). The results of the test case study analysis as well as the positive feedback from the involved practitioners indicate, iRBA is the most promising and practical risk and prioritization

[1] http://www.pairwise.org.

methodology to date. Its quality is far superior compared to other methods, and it makes the entire SDLC and testing process much more effective. Unlike other non-iRBA methods, iRBA is agnostic framework applicable to any project, environment and SDLC model. In addition, the results indicate risk prioritization of multiple applications and early detection of major interoperability issues are indeed feasible and much more effective by iRBA.

iRBA made a significant revolution in the field of software engineering by shortening product time to market and improving project predictability, cost, quality and efficiency.

11.7 Future Work

The consistent and sustained delivery of results and solutions by using iRBA is testimony to the effectiveness of this intelligent methodology. These activities were essential to fulfill the vision and bring it to reality. To the best of our knowledge, this kind of holistic methodology that covers most of the software testing challenges and maximizes overall efficiencies has not yet been done.

This chapter about the iRBA is the perfect marriage between iRBA theory and practice, as well as a valuable source of insight and ideas. It can inspire researchers and scientists to learn iRBA methodology and its mindset, to propose and apply more powerful and more accurate models for software engineering challenges. The software engineering community can replicate the consistent success to achieve similar significant improvements by adopting iRBA to improve products validation quality and ensure success in the competitive marketplace of software acceleration.

Despite this success and recent progress, there are still some directions for future work in this area. It is recommended to use AI algorithms to learn the impact level of the iRBA-Link on the impacted test cases cluster in the short and long terms, and update the policy to update their properties respectively, e.g. strong link, average link, basic link, weak link, or false link.

Additional future work could be edge cases, like very long or a complex high priority test case that might block the execution progress of other test cases. iRBA is also coping with these kinds of cases, but that is out of the scope of this chapter. It is recommended to formulize these edge cases and their specific iRBA-Reward-Function.

Acknowledgments

This work was funded by Intel Development Center Jerusalem.

References

[1] A. R. Lenz, A. Pozo, and S. R. Vergilio, Linking Software Testing Results with a Machine Learning Approach, Engineering Applications of Artificial Intelligence, vol. 26, no. 5–6, pp. 1631–1640, 2013.

[2] B. Busjaeger and T. Xie, Learning for Test Prioritization: An Industrial Case Study, in Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016, pp. 975–980.

[3] Bauer, T., Stallbaum, H., Metzger, A., Eschbach, R.: Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest. Softw. Eng. 121, 99–111 (2008).

[4] C. Anderson, A. V. Mayrhauser, and R. Mraz, On the Use of Neural Networks to Guide Software Testing Activities, in International Test Conference, 1995, pp. 720–729.

[5] Catal, C. (2011). Software fault prediction: A literature review and current trends, Expert Systems with Applications 38, 4, pp. 4626–4636.

[6] D. Graham, E. Veenendaal, I. Evans, R. Black. Foundations of Software Testing ISTQB Certification, International Software testing Qualifications Board, 2010.

[7] Felderer, M., Ramler, R.: Integrating risk-based testing in industrial test processes. Softw. Qual. J. 22(3), 543–575 (2014).

[8] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In Proc. ICSM 1999, pp. 179–188.

[9] Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold, and IEEE Computer Society. 2001. Prioritizing Test Cases For Regression Testing. IEEE Transactions on Software Engineering 27, 10 (2001), 929–948.

[10] H. Hemmati and F. Sharifi, Investigating NLP-Based Approaches for Predicting Manual Test Case Failure, 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018.

[11] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, Reinforcement learning for automatic test case prioritization and selection in continuous integration, in Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2017, pp. 12–22.

[12] J. Pollack, J. Helm, and D. Adler, "What is the iron triangle, and how has it changed?" Int. J. Manag. Projects Bus., vol. 11, no. 2, pp. 527–547, 2018.

[13] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In Proc. ICSE 2002, pp. 119–129.

[14] James Bach, Heuristic Risk-Based Testing, Software Testing and Quality Engineering Magazine, November 1999, pp. 96–99.

[15] Khatibsyarbini, M., Isa, M. A., Jawawi, D. N., & Tumeng, R. (2017). Test

case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology.

[16] Kloos, J., Hussain, T., Eschbach, R.: Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In: 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 26–33. IEEE (2011).

[17] Koch, P., Schekotihin, K., Jannach, D., Hofer, B., and Wotawa, F. (2019). Metricbased fault prediction for spreadsheets, IEEE Transactions on Software Engineering.

[18] Luo, G., Bochmann, G., Petrenko, A.: Test selection based on communicating non-deterministic finite-state machines using a generalized Wp-method. IEEE Trans. Softw. Eng. 20(2), 149–162 (1994).

[19] Martin Fowler and M Foemmel. 2006. Continuous integration. (2006).

[20] Moran, A. 2015. Managing Agile: Strategy, Implementation, Organisation and People. Springer Verlag, Heidelberg, NY.

[21] P. M. Duvall, S. Matyas, and A. Glover. 2007. Continuous Integration: Improving Software Quality and Reducing Risk. Pearson Education.

[22] P. Tonella, P. Avesani, and A. Susi, Using the Case-Based Ranking Methodology for Test Case Prioritization, in IEEE International Conference on Software Maintenance, 2006, pp. 123–133.

[23] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, System-Level Test Case Prioritization Using Machine Learning, 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), 2016.

[24] Richard S. Sutton and Andrew G. Barto. 1998. Reinforcement Learning: An Introduction (1st ed.). MIT press Cambridge.

[25] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, Using Semi-supervised Clustering to Improve Regression Test Selection Techniques, in IEEE International Conference on Software Testing, Verification and Validation, 2011, pp. 1–10.

[26] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In Proc. FSE 2014, pp. 235–245.

[27] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In Proc. ESEC/FSE 2011.

[28] Seol, S., Kim, M., Chanson, S.T., Kang, S.: Interoperability test generation and minimization for communication protocols based on the multiple stimuli principle. IEEE J. Sel. Areas Commun. 22(10), 2062–2074 (2004).

[29] Singh, R., Livshits, B., and Zorn, B. (2017). Melford: Using neural networks to find spreadsheet errors, Microsoft Technical Report MSR-TR-2017-5.

[30] Stale Amland, Risk Based Testing and Metrics: Risk analysis fundamentals and metrics for software testing including a financial application case study, The Journal of Systems and Software, Vol. 53, 2000, pp. 287–295.

[31] Stallbaum, H., Metzger, A.: Employing requirements metrics for automating early risk assessment. In: Wsh. Measuring Requirements for Project and Product Success (MeReP), pp. 1–12 (2007).

[32] T. Hall and D. Bowes, The State of Machine Learning Methodology in

Software Fault Prediction, 2012 11th International Conference on Machine Learning and Applications, 2012.

[33] Wendland, M.F., Kranz, M., Schieferdecker, I.: A Systematic approach to risk-based testing using risk-annotated requirements models. In: 7th International Conference on Software Engineering Advances (ICSEA), pp. 636–642 (2012).

[34] V. Garousi, R. Özkan, A. Betin-Can, Multi-objective regression test selection in practice: An empirical study in the defense software industry, Inf. Softw. Technol., vol. 103, pp. 40–54, Nov. 2018.

[35] Y. Yang, X. Huang, X. Hao, Z. Liu, and Z. Chen, An Industrial Study of Natural Language Processing Based Test Case Prioritization, 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017.

[36] Y. Ledru, A. Petrenko, S. Boroday, N. Mandran, Prioritizing test cases with string distances, Automated Software Engineering 19 (1) (2012) 65–95.

[37] Zimmermann, F., Eschbach, R., Kloos, J., Bauer, T.: Risk-based statistical testing: a refinement-based approach to the reliability analysis of safety-critical systems. In: 12th European Workshop on Dependable Computing (EWDC) (2009).

https://doi.org/10.1142/9789811239922_0012

Chapter 12

A Qualitative Reasoning Model for Software Testing, based on Combinatorial Geometry

Spyros Xanthakis[a] and Emeric Gioan[b]

[a] *Athens University of Applied Science, Athens, Greece*
[b] *LIRMM, University of Montpellier, CNRS, Montpellier, France*

12.1 Introduction

Let us first give a short summary of the paper. We introduce an operational *qualitative model* for numerical algorithms. This model enables *qualitative algorithmic reasoning* during the unit testing process, supported by an automatic generation of boundary test data. It contains the spatial encoding of all the functionally equivalent regions (called *homodromies*). The model, viewed as a qualitative abstraction of the algorithm, is automatically generated thanks to multiple targeted executions of its instrumented basic conditions, that permit the interpolation of a set of linear equations and their corresponding *hyperplanes*. All feasible paths (when all basic conditions are linear) are identified in the form a *Ternary Decision Tree*. Each feasible execution path corresponds to a geometric region supported and/or delimited by a finite set of hyperplanes of the vector space. *Oriented Matroids* allow the encoding of the relative positions of all such regions of various dimensions, in a purely combinatorial way, using a finite signed set system, supported by a rich mathematical theory. This model uses a *qualitative space algebra* and enables *qualitative reasoning*: regions are identified according to qualitatively valued inputs, which are, in their turn, propagated through them. The proposed model permits a global/local envisionment

of the algorithmic behaviour in an abstract level. The cocircuits of the underlying oriented matroid play the role of testing building blocks: the combination of their numerical coordinates enables the automatic generation of limit test data that lies at the boundary of any critical surface of any dimension.

Let us now present the paper and its context in more details. From the very beginning, the software engineering community tried to adopt artificial intelligence (AI) techniques for assisting an especially time-consuming and complex phase of the software development process: unit testing. *Unit testing* is destinated to assess the units programmed during the implementation phase (a C-procedure, a Python method). During this phase, the tester, who is often the developer, has to design test data (according to unit testing criteria), prepare the right execution environment, execute the (possibly instrumented) software, filter the outputs, analyse and validate the final results. Testing criteria could be functional, like, "Test what happens when alarm is on", "Check the answer of the system when the temperature is very high", or structural like, "Check what happens if a loop statement is not executed", or "Be sure that, at the end of unit testing, all statements have been executed at least once", etc.

Evolutionary (or *search based*) techniques, initially devoted to the AI area, have nowadays proved their efficiency in software testing, and particularly in the automatic test data generation. A first application of evolutionary techniques for unit testing was proposed by one of the authors of this paper [1, 2]. It consisted in expressing a structural test data objective as an optimization problem handled by a conventional steady state genetic algorithm (GA) coupled with an elementary data flow analysis. Many works followed that improved the performances and extended the range of techniques and applications; they are now grouped under the name *evolutionary testing* [3–8] which is no more limited to the use of GAs but encompasses a wide spectrum of AI techniques. More broadly, evolutionary (or search based) software engineering, can now tackle with different testing tasks: functional testing, integration testing, mutation testing, regression testing and test case planning [5, 9–13]. Test data generation, and especially limit (or boundary) test data, constitute a major open issue in the software development process. Our testing data generation philosophy does not rely on a specific structural coverage objective (i.e. cover all the statements), nor uses an a priori testing model (i.e. a state machine or a control graph), nor a random/heuristic/evolutionary technique. Under certain assumptions

that will be presented in the beginning of the next section (briefly, for a constant-time algorithm based on linear inequalities), it handles, formally and exhaustively, all the feasible execution paths of the source code.

In our view, existing automatic testing generation techniques, do not dispose of a sufficient high level of abstraction to assist testers intelligently. The testing process cannot be reduced to an input/output comparison task. It often involves designing activities (design of the testing strategy and its deployement, redesign a defectuous code, etc.) but also reverse engineering tasks (assisted by many visualisation and/or debugging tools, etc.). All those activities necessitate a minimum of understanding and reasoning about the actual and/or the required behaviour of an algorithm. We will try to illustrate our point by a (non exhaustive) list of some questions that may arise during testing (assuming that inputs are endowed with an absolute order relation). Questions of this kind are answered in Sec. 12.4.

- How and in what direction a specific output is impacted by a small change of inputs? Which are the inputs that are influencing a condition in the source code? In what direction could one change the inputs in order to switch its truth value from `False` to `True`?
- What happens at a close neighborhood of a critical point? If the temperature is very high and the pressure is normal will the alarm be triggered? Will it stay triggered even if the temperature raises "indefinitively"? Is it possible to find contiguous regions with different behaviour, and, if yes, what are the limit test data that separate them?
- If one decreases the element of an array or if this element is very small, will it eventually appear at the beginning of an ascending order sorted array? In the Knap Sack problem if one decreases the weight of an already selected item, will the total value increase or decrease? In the Traveler Salesman Problem, if the distance between two cities is decreased, will their orders be closer in the solution?

We argue that all those kind of questions are pervasive in software testing and contain indisputably qualitative reasoning aspects, extensively studied by the qualitative reasoning community in AI, such as *comparative analysis and envisionment*, *order-of-magnitude* and *spatial reasoning* [14–20]. We will illustrate that spatial properties are implicit when reasoning about numerical algorithms. We do not claim that qualitative reasoning exhausts all the mental activities of unit testing (and even less,

of algorithmic reasoning in general), but that it constitutes an essential component.

In our approach, an algorithm is viewed as a *fragmentation* process of the input space considered as a real vector space. It uses sequences of nested and iterated logical conditions in order to fragment the input space into small connected pieces, the fragments. At each point (which corresponds to a specific input) of a fragment is associated a uniform sequence of computations. One of the main tasks of software testing could be compared to the task of checking whether those fragments are well "glued" together. Our central working assumption can thus be formulated as follows: if one is capable, first, of qualitatively expressing the underlying spatial information of the fragments (i.e., contiguity, direction, intersection, inclusion, etc.) and, secondly, qualitatively expressing input values (i.e., small, big, close to, etc.), while ensuring their propagation between and inside those regions, then an algorithmic qualitative reasoning is possible.

We present a qualitative reasoning algorithmic model. Its construction is illustrated in Fig. 12.1. The spatial relations between fragments are expressed using a very active area of combinatorial geometry: Oriented Matroid theory (whose fundamentals are recalled in Sec. 12.3). Qualitative propagation is ensured by an order-of-magnitude algebra. To any cocircuit of the oriented matroid (combinatorial encoding of a 1-dimensional region) is naturally associated the quantitative information (real coordinates) of its location in the space. This is used to produce, automatically, by means of algebraic combinatorial properties, limit test data lying at the boundary of any critical surface of any dimension. Our model necessitates only a simple instrumentation of the basic conditions of the source code. It can be then used for functional/non-functional as well for object-oriented/non-object-oriented languages ranging from assembly language to Fortran, C, LISP, C++, Java, Python, etc. It encompasses all applications using mainly numeric inputs, such as avionics, statistics, banking, simulation, robotics, etc. Further comments are given in the conclusion.

12.2 What the model contains

Our model is based on an instrumented source code. In the whole paper, our assumption is that the source code implements a constant-time algorithm, and that it is based on real (decimal) linear (in)equalities, so that all possible conditions executed for all possible inputs amount to evaluate some real linear functions of the input vector space, yielding a finite set of

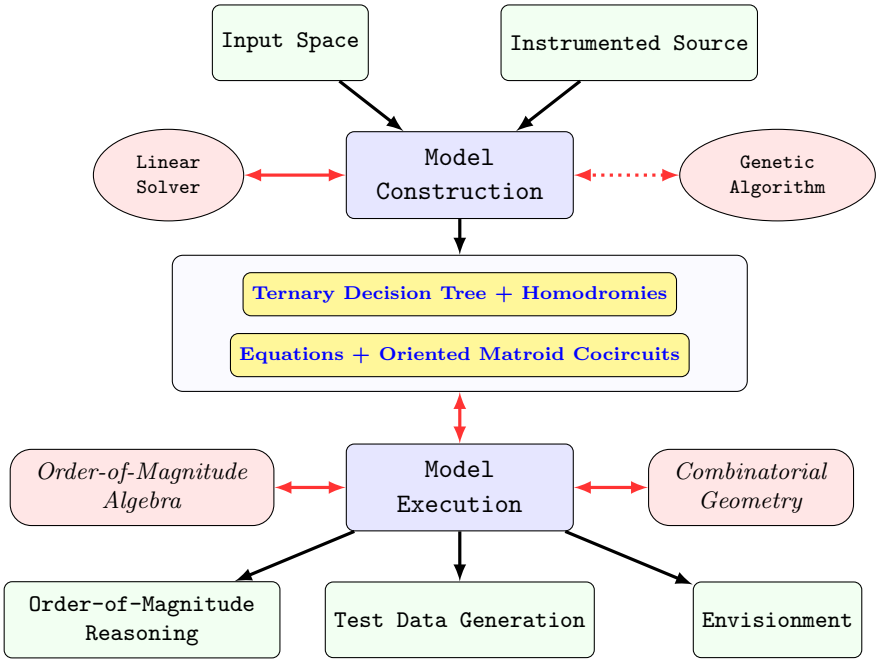


Fig. 12.1 The two main steps of the algorithmic qualitative modelling process: `Model Construction` and `Model Execution`. The *Instrumented Source* code, and its *Input Space* are given. In case of non-linearity a *Genetic Algorithm* can be used to detect a critical point. A rational *Linear Solver* builds a *Ternary Decision Tree* and interpolates the equations (generated by the basic conditions of the source code), their corresponding oriented matroid cocircuits and all the homodromies (geometric regions associated to feasible paths). `Model Execution` will then use *Order-of-Magnitude Algebra* combined with a *Combinatorial Geometry* algebra (playing the role of a qualitative space algebra) enabling the abstract *Execution* and *Envisionment* of the source code behaviour. Finally, quantitative information associated with the cocircuits permits the automatic generation of limit test data that lies exactly at the boundaries (of any dimension) of critical surfaces.

useful real linear functions, as detailed below. In this case, the model is completely well-defined and extensive. In the non-linear case (and in the non-constant-time case), that will not be addressed in this paper, the same technique can be used locally, at a close neighbourhood of a critical point of the input space, by considering appropriate linear inequalities, possibly with an approximation (this can be based on the Jacobian matrix, and/or found by Genetic Algorithms techniques).

We first introduce the content of our model (Sec.12.2.1). Next, we illustrate it on a simple example that will serve as running example throughout this paper. Section 12.2.3 details the technical construction of the model.

12.2.1 *Conditions, Equations, Paths, Super-covectors, and Homodromies*

Let us start with a source code using numeric linear (in)equalities as conditions. When the source code is executed for a given input, a sequence of conditions is executed. We say in that case that these conditions are *sensitized*. To each condition corresponds a truth value, `True` or `False`, observed during the execution path, with respect to the current variable values. We will not focus on the sequence of truth values but on a sequence of signs $+$, $-$, or 0, which we call the *sign-path*, defined as follows. When the condition is executed in the source code, written in the form [`left-hand-side (LHS) comparison-operator right-hand-side (RHS)`], then we record the sign of the value of [`LHS` $-$ `RHS`] (with respect to the current variable values). Intuitively speaking, when the condition is an inequality, the 0 case can be thought of as a "limit case between `True` and `False`". Our model takes into account these limit cases and handles them in a precise geometric combinatorial structure.

According to our assumption, each condition in the sequence consists to checking whether an associated real linear function of the input space has a positive, negative or null value. In what follows, we abusively call this linear function an *equation* (though it is the underlying inequality that is used). Generally, equations do not appear explicitly in the source code (in contrast with our toy example), and the same source condition may yield several equations (in the case of a loop, typically). The set of all resulting equations for all possible executions forms a set E, which is finite (because of the constant-time assumption). It is computed by a tree search based on the sign-path associated to an interpolation technique using a linear solver (see Sec. 12.2.3). We call *equation-path* the sequence of sensitized equations (among E), and with each equation is associated a sign in {`+`, `-`, `0`} according to the value of its corresponding linear function. A supplementary equation "at infinity" is added when the initial inequalities are affine, in order to use a standard vector space setting; its sign will always be `+` for executions; it will also permit to model the algorithm's behavior "at infinity". We stress the fact that two inputs admit the same sign-path if and only if they admit the same equation-path (though the two sequences of signs can be different). This property ensures that, for each execution path, a unique equation (with a unique sign) will be associated with every basic condition, thus permitting its interpolation.

Finally, each possible execution is encoded by associating each element

of E with either its sign from the equation-path or the *undetermined sign* $\times$ when the equation is not sensitized in the equation-path, yielding what we call a *super-covector* over E, or *scovector* for brevity. Those combinatorial objects can actually be understood as parts of the oriented matroid structure underlying the set E of real equations in the real input vector space, as detailed in Sec. 12.3. Note that two inputs admitting the same scovector do not generally yield the same output (as in our toy example); however they sensitize the same execution path. The converse is not true: two inputs sensitizing the same execution path may admit distinct scovectors (because of the `0` sign of conditions, as noted above). The resulting scovectors yield equivalence classes aggregating inputs that sensitize the same execution path.

The geometric interpretation of the previous equivalence relation is captured by the concept of *homodromy*. Homodromies — from the greek, $\acute{o}\mu o\iota o\varsigma(same)$, $\delta\rho\acute{o}\mu o\varsigma(path)$ — are connected regions of the (geometrized) input space, that exhibit the "same" algorithmic behaviour. They group inputs that are operationally treated in the "same manner". Clearly, homodromies correspond to feasible execution paths and can be compared to a sort of "software phases". Two inputs belong to the same homodromy (we say in that case that they are *homodromic*) if one can connect them with a "continuous" geometric path without changing the sign-path (or, equivalently, the equation-path). As explained above, homodromies are combinatorially encoded by scovectors, and we shall say that the homodromy is the *region* of a scovector encoding the homodromy as above. Let us emphasize this notion as it constitutes the central notion of this work. (Let us mention that the geometric concept of homodromy as defined below could be extended to any source code based on a geometric input space.)

Definition 12.1. We call *homodromies* the path-connected components (of the input space) induced by the equivalence relation based on the sign-path (or, equivalently, the equation-path). Homodromies form a partition of the whole input space called the *fragmentation* of the input space with respect to the source code (see Definition 12.6 below for a combinatorial counterpart).

12.2.2 *Running example*

Suppose that one wishes to perform structural dynamic testing of the following source code, which implements the specifications of an approximative and simplified model of the water transition phase diagram from classical

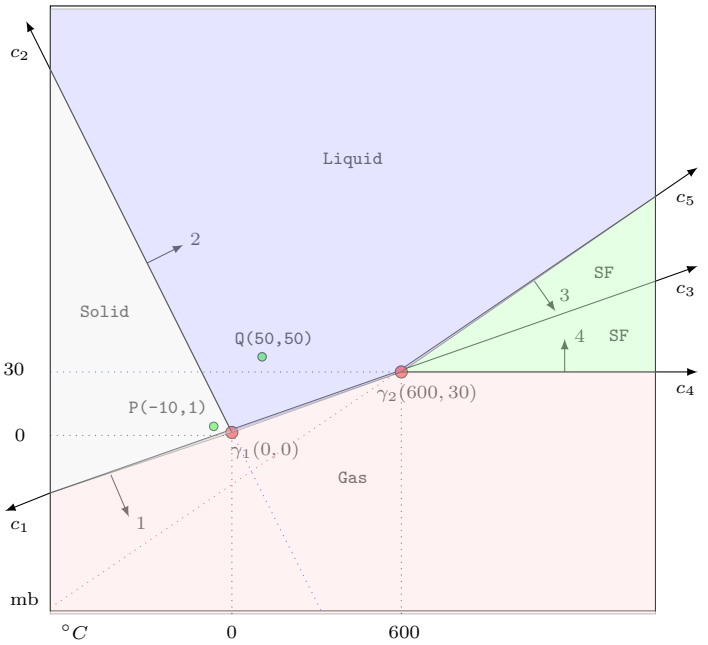


Fig. 12.2 Phase Transition Diagram of the toy algorithm. Four equations, induced by the conditions in the source code, delimit four zones corresponding to distinct outputs (phases). These equations $e_1, \ldots, e_4$, are depicted as 1, 2, 3, 4, and "oriented" by arrows towards their positive side (the points where the equation is positive). The boundary of the `Solid` zone (formed by regions/homodromies yielding the same `Solid` output) is formed by the equations e_1 and e_2 sensitized by the internal point P(`-10,1`). Similarly, the point, Q(`50,50`) (`Liquid` zone) sensitizes its border equations e_1, e_2, e_3. The points γ_1 and γ_2 are intersections of lines corresponding to the equations, and c_1, c_2, c_3, c_4, c_5 denote the useful directions of those lines at infinity.

physics, as illustrated in Fig. 12.2. The `WaterPhases` algorithm takes two decimal inputs: t (between -100.0 and $+1000.0$) and p (between -100.0 and $+100.0$), representing the temperature (in degree Celsius) and the pressure (in bars) of the water, respectively. Then, it outputs its state: `Solid, Liquid, Gas, SuperFluid`.

```
if t >= 20*p:
      if p > 30: return 'SuperFluid'
      else: return 'Gas'
elif 10*t + p <= 0: return 'Solid'
elif t - 10*p < 300: return 'Liquid'
else: return 'SuperFluid'
```

The previous source code uses four basic conditions: `(t >= 20*p)`, `(10*t + p <= 0)`, `(t - 10*p < 300)` and `(p > 30)`. Basic conditions (simply called *conditions*) are expressions separated by a comparison operator {`<=`, `>=`, `==`, *etc.*}. They are uniquely identified by their textual position inside the source code and may appear in many different forms, like, inside compound conditions using logic connectors, and/or at the right hand of boolean assignment statements. Note that, the (basic) conditional statement "`if (t >= 20*p)`" is equivalent to "`if (t - 20*p >= 0)`".

Let's now execute the previous code for the input $t =$ `-10.0` and $p =$ `1.0`, corresponding to a cold temperature of $-10°C$ and a normal atmospheric pressure of 1 *bar*. First condition, `t >= 20*p`, is executed (*sensitized*) yielding the truth value `False` with the negative value $t - 20 * p = (-10) - 20(1) = -30$. The *execution path* follows the `else` alternative and sensitizes the condition `p + 10*t <= 0` which is `True` (since `1+10.(-10)=-99<=0`) and the observed value is negative; the final output will be the `Solid` (ice) phase. In conclusion, the two conditions have been sensitized, yielding the sign-path `[--]`. These conditions can be interpolated (as it will be explained in Sec. 12.2.3) yielding respectively the linear equations $e_1(t, p) = t - 20p$ and $e_2(t, p) = 10t + p$. The *equation path* associated to the point $P(-10, 1)$ will be thus noted 1^-2^-. The resulting *scovector* is [`--xx+`] (the last `+` sign comes from an equation added "at infinity" as mentioned above).

Table 12.1 includes these results for all executions/homodromies of the toy example.

12.2.3 *How the model is built technically*

In this section, we present how the successive executions of the source conditions permit the interpolation of their corresponding (unique) equation inside each homodromy. This is accomplished by a progressive expansion of a Ternary Decision Tree.

Ternary Decision Trees. Let $\mathbf{E} = \{e_i\}_{i=1..n}$ be a set of linear equations on $\mathbb{R}^r$. A *Ternary Decision Tree (TDT)* is a tree where any non terminal node belongs to $\mathbf{E}$ and admits 1, 2 or 3 successors (according to their feasibility) connected with *sign labels* in {`+`, `-`, `0`} (each sign appearing only once). For convenience, one can collapse identical brother subtrees in a unique subtree decorated by the union of the original signs (see Fig. 12.3).

Table 12.1 The 11 homodromies of the toy algorithm (grouped according to their topological dimension), denoted $\Omega_1, \ldots, \Omega_{11}$, their sign-paths and equation-paths, their resulting encoding as scovectors, the corresponding output, and an illustrative boundary test data proposed by the tool (see Sec. 12.4.3). The column "Test data" gives an input yielding the corresponding sign-path. For instance, in the last row, the test data `[600,30]` means that putting the temperature at 600 degrees and the pressure at 30 bars, the algorithm output (the water phase) will be `Gas` and the sensitized homodromy corresponds to the scovector [`0xx0+`].

Ω_i	*Sign-path*	*Equation-path*	*SCovector*	`Output`	`Test data`
1	[--]	1^-2^-	[--xx+]	`Solid`	`[-100,999]`
2	[-+-]	$1^-2^+3^-$	[-+-x+]	`Liquid`	`[599.99,30]`
3	[-++]	$1^-2^+3^+$	[-++x+]	`SuperFluid`	`[600.3,30.02]`
4	[++]	1^+4^+	[+xx++]	`SuperFluid`	`[600.21,30.01]`
5	[+-]	1^+4^-	[+xx-+]	`Gas`	`[-100,-100]`
6	[-0]	1^-2^0	[-0xx+]	`Solid`	`[-100,1000]`
7	[-+0]	$1^-2^+3^0$	[-+0x+]	`SuperFluid`	`[600.1,30.01]`
8	[0+]	1^04^+	[0xx++]	`SuperFluid`	`[600.2,30.01]`
9	[+0]	1^+4^0	[+xx0+]	`Gas`	`[600.01,30]`
10	[0-]	1^04^-	[0xx-+]	`Gas`	`[-100,-5]`
11	[00]	1^04^0	[0xx0+]	`Gas`	`[600,30]`

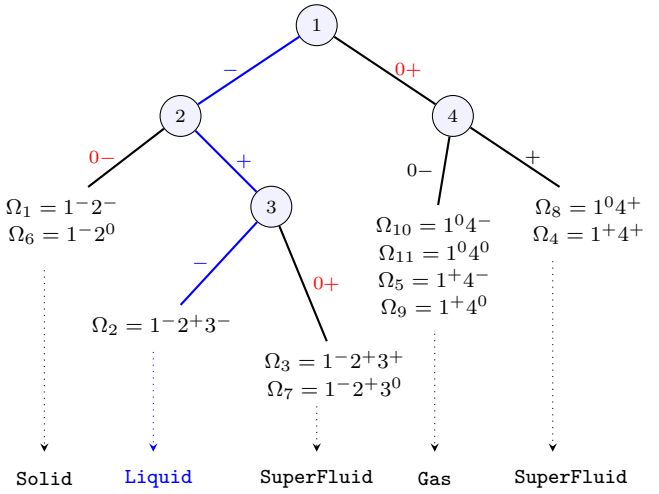


Fig. 12.3 The Ternary Decision Tree (TDT) modelling the algorithmic behaviour of the `WaterPhases` algorithm. Each of the 11 leaves corresponds to a unique homodromy of Table 12.1. The equation-path $\varrho = 1^-2^+3^-$ sensitized by Q(`50,50`) corresponds to the `Liquid` homodromy Ω_2. Note that those equations $1, 2, 3$ form its geometric boundary (see Fig. 12.2). As addressed in Sec. 12.2.3, this equation-path ϱ admits 6 tree alternatives: $\{1^0, 1^+, 1^-2^0, 1^-2^-, 1^-2^+3^0, 1^-2^+3^+\}$. A rational linear solver will produce solutions (or not) for those alternatives, and thus new points to be executed that will eventually sensitize the 4th equation.

Each *leaf* (node with no successor) expresses a geometric region, i.e. a single homodromy, thus a feasible execution path, labeled by its equation-path. A tree using n equations will admit at most 3^n feasible leaves. Their feasibility depends on the intersection pattern of the equations. (Let us mention that this feasability is encoded in the underlying oriented matroid.)

Tree Expansion Algorithm. The tree expansion algorithm follows a conventional tree breadth-first strategy (BFS). At a first stage, a random sampling provides a bootstrap point. Consider for instance that the point $Q = (50,50)$, of the `Liquid` homodromy Ω_2, is the bootstrap point. Q has a sign-path [-+-] and sensitizes equations $\{e_1, e_2, e_3\}$. Using the rational linear solver, in a small enough neighborhood of Q, it is now possible to generate a sufficient number of homodromic points enabling their interpolation. In fact, since all points belong to the same homodromy, we know that the condition `(t >= 20*p)` expresses a unique equation of the form: $e_1(t,p) : k_1 t + k_2 p + k_3 = 0$ (since we use two inputs). In our example, the execution of two additional homodromic points, yields two more values for the equation e_1, and allows the evaluation of the coefficients k_1, k_2, k_3, and thus the exact mathematical expression of $e_1(t,p) = t - 20p$. The same interpolation is done by observing the values of the conditions `(10*t + p <= 0)` and `(t - 10*p < 300)`, yielding the equations $e_2(t,p) = 10t + p$ and $e_3(t,p) = t - 10p - 300$. The equations having been identified, it is now possible to reconstitute the first equation-path path of the TDT: $\varrho = 1^-2^+3^-$ (see the path in Fig. 12.3), yielding the scovector [-+-x+]. At a second stage, follows the generation of the 6 *tree alternatives* of ϱ: $\{1^0, 1^+, 1^-2^0, 1^-2^-, 1^-2^+3^0, 1^-2^+3^+\}$ (see Fig. 12.3). Each tree alternative corresponds to a subpath of ϱ starting from the root, where the last node has been extended with only one branch different from the original one. Each alternative corresponds to a system of (strict) linear (in)equalities submitted to a rational linear solver. Starts now, a classic iterative process of a ternary tree expansion. New solutions and sign alternatives will be found, checked, rejected or added as new leaves.

Note that equalities can be handled in the same way as inequalities (consistently with the arbitrary choice of signs used for the sign-path). Thus, we shall not make the distinction with inequalities anymore. Once the TDT has been built, one thus disposes of a set E of linear inequalities, with which can be associated an oriented matroid.

12.3 Which mathematical results will be needed

Oriented matroid theory is a rich mathematical theory born in the 1970's as a combinatorial abstraction of linear algebra, convex geometry, and graph theory. In this paper, we focus on a small portion of this theory, that is, on vectorial oriented matroids and how they encode regions of all dimensions in real hyperplane arrangements. General oriented matroids are both combinatorial objects, by means of combinatorial axioms, and topological objects, by means of pseudosphere arrangements; see [21]. Section 12.3.1 is a short introduction to oriented matroids theory with some classical results that are necessary for our formal setting. Next, we will build on it for the sake of our model. We will give no technical details nor formal proofs for our constructions in this paper; see [22] for further deepening.

12.3.1 *Preliminaries on Oriented Matroid theory*

General setting. Let us start with a set of n linear inequalities in a vector space or an affine space (see Sec. 12.2). In the latter case, in a standard way, we embed the affine space into a vector space with one more dimension, corresponding to one more variable. The affine space corresponds to this variable having a value equal to 1. And we add the inequality (considered as being *at infinity*) where the additional variable is greater than zero. From now on, we thus assume that we are given a set of n inequalities in a real vector space (possibly including one inequality at infinity). Denoting the variables by x_i, $1 \leq i \leq r$, each inequality $\sum_{1 \leq i \leq r} \alpha_i.x_i \geq 0$ can be seen as a vector $(\alpha_1, \ldots, \alpha_r)$ of the ambient space. Up to considering a smaller ambient space, we can assume that these n vectors span the ambient space $\mathbb{R}^r$, and we will always make this assumption, which is crucial for further results and geometric representations.

We also assume that none of these vectors is the null vector. Hence, each induced linear equation $\sum_{1 \leq i \leq r} \alpha_i.x_i = 0$, yields an hyperplane of the ambient space (that is, a $(r-1)$-dimensional subspace), which is the set of vectors $(x_1, \ldots, x_r)$ satisfying the equation.

Combinatorial encoding in terms of oriented matroids. Accordingly to the previous setting, one disposes of a finite ground set $E = \{e_1, ..., e_n\}$ of a set of (non-null) vectors of the real space $\mathbb{R}^r$ (spanning the ambient space), possibly with repetitions of the same vector with different indices. Each vector $e \in E$ provides the equation $e.x = 0$, for $x \in \mathbb{R}^r$,

of a hyperplane H_e of $\mathbb{R}^r$ called *the hyperplane of* e, and provides the inequalities of two half-spaces $e.x > 0$ and $e.x < 0$, for $x \in \mathbb{R}^r$, called *the positive and negative half-space of* e, denoted H_e^+ and H_e^-, respectively. The hyperplane H_e can be equally denoted H_e^0. Also, the element at infinity will be denoted p, when it exists. Such a set of hyperplanes is called an *hyperplane arrangement.*

Those hyperplanes H_e and half-spaces H_e^+ and H_e^-, for all $e \in E$, subdivide the ambient space $\mathbb{R}^r$ into cells of various dimensions. For the sake of a geometric representation of these cells, notably used in the figures of this section, we actually consider a central unit sphere S^{r-1} of $\mathbb{R}^r$ as the ambient object, and the spheres S^{r-2} in S^{r-1} which are the intersections of the hyperplanes with S^{r-1}. When there is an element at infinity p, then, by symmetry, we can consider only the half-sphere S^{r-1} on the positive side of p, and we represent p as a sphere at infinity bounding the figure. In this way, the cells of $\mathbb{R}^r$ defined by the hyperplanes and half-spaces canonically correspond to cells of S^{r-1}, except the null vector of $\mathbb{R}^r$ which is not represented. See Fig. 12.4.

A *signed-set* C on E is defined by giving a sign in $\{+, -, 0\}$ to each element of E. The subset of elements with a $+$, $-$, or 0 sign is denoted by C^+, C^- or C^0, respectively. Thus, a signed-set yields a partition $E = C^+ \cup C^- \cup C^0$. The sign of $e \in E$ in the signed-set C is denoted C_e. The subset $\underline{C} = C^+ \cup C^-$ of E is called the *support* of C. For $A \subseteq E$, the signed-set $C \setminus A$ on $E \setminus A$ is defined to have $C^+ \setminus A$ as a positive part and $C^- \setminus A$ as a negative part. For example, writing elements of $E = \{1, \ldots, 5\}_<$ in a list, $C = [\texttt{---0+}]$ is the signed-set with $C^+ = \{5\}$, $C^- = \{1, 2, 3\}$ and $C^0 = \{4\}$.

For $x \in \mathbb{R}^r$, we define the *covector* C *associated with* x as the signed-set defined, for each $e \in E$, by giving a 0, $+$ or $-$ sign to e whether x belongs to the hyperplane, the positive half-space or the negative half-space of e, respectively. Formally, we have:

$$C_e = \text{sign}(e.x).$$

The (finite) set of all covectors associated with all $x \in \mathbb{R}^r$ is denoted by $\mathcal{C}\text{ov}$ and forms the family of covectors of an *oriented matroid* M on E. Formally, an oriented matroid is a finite ground set E provided with a family of signed-sets on E called its *covectors* and satisfying some combinatorial axioms. Here, the oriented matroid is called *vectorial* as it was built from vectors. Actually, an oriented matroid can be characterized by different families of signed-sets with different axiom systems, such as its cocircuits

or its topes addressed below. As E spans a vector space of dimension r, we say that the *rank* of M is r. The cells of the hyperplane arrangement are in canonical bijection with the covectors. Covectors combinatorially describe the relative positions of the cells. We call *tope* a maximal covector, that is, a covector not containing the zero sign. Topes correspond to *full-dimensional cells* of the space delimited by the hyperplanes of E (that is, to connected components of the complementary set in $\mathbb{R}^r$ of the union of the hyperplanes). We call *cocircuit* a minimal non-null covector. Geometrically speaking, cocircuits correspond to 1-dimensional cells. In a sphere representation, they correspond to the *points* obtained as intersections of spheres representing the elements of E.

The notions described above are illustrated in Fig. 12.4. For instance, covectors with no - sign correspond to cells bounding the cell encoded by [+++++] (see [0++++], [++++0], [+++0+], [0+++0], [+++00], and [0+00+]), topes with one - sign correspond to full-dimensional cells reached from the previous cell by crossing one hyperplane (see [−++++] and [+++−+]).

Conformal composition. Two covectors $C, D \in \mathcal{C}\text{ov}$ are called *conformal* (to each other) if $C^+ \cap D^- = C^- \cap D^+ = \emptyset$, that is, if no element $e \in E$ has a different non-zero sign in C and D. In other words, the corresponding cells are in the boundary of the same full-dimensional cell.

Given two conformal covectors C and D, the covector obtained by *conformal composition of C and D* is the covector denoted $C \circ D$, or $D \circ C$, whose positive elements are $C^+ \cup D^+$ and whose negative elements are $C^- \cup D^-$. Geometrically, when the corresponding cells are not comparable for inclusion, $C \circ D$ corresponds to the cell given by the strict convex hull of the cells corresponding to C and D. The next result will be crucial for us.

Theorem 12.1. *Every covector is the result of the conformal composition of the set of cocircuits that are conformal to it.*

The geometric interpretation in terms of full-dimensional cells is simply that every open bounded convex polytope (tope) is the strict convex hull (conformal composition) of its extremal points (cocircuits conformal to the tope). So, the above property can be seen as the combinatorial counterpart of this classical property in convex geometry. For instance, in Fig. 12.4, we have that [+++++] = [0+00+] ∘ [+++00] ∘ [0+++0], and we have that [−+0++] = [0+00+] ∘ [−+0+0].

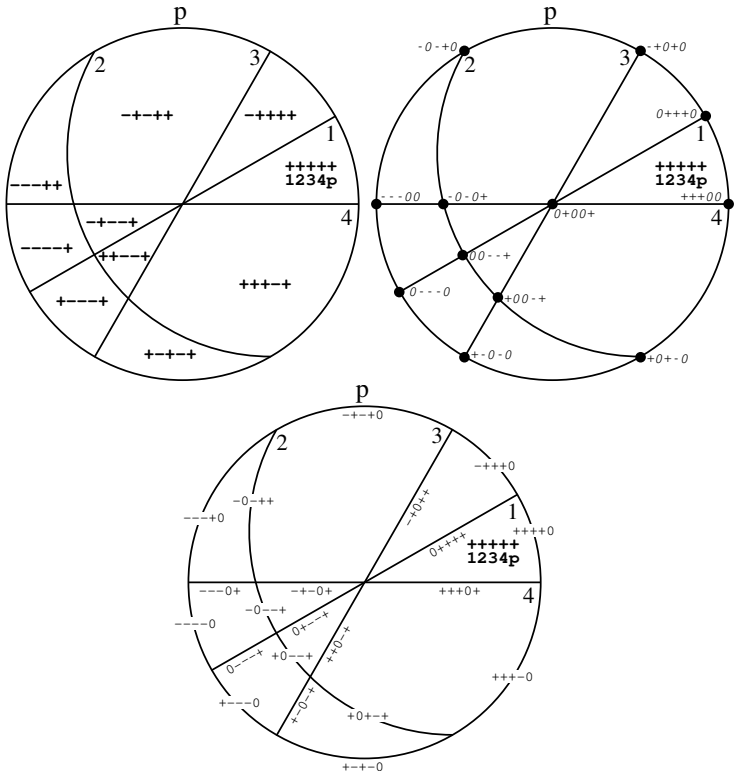


Fig. 12.4 Oriented matroid covectors encoding cells of the hyperplane arrangement, in a sphere representation with the element p at infinity. The signed-sets are indicated as lists with respect to the ordering $E = \{1, 2, 3, 4, p\}_<$. The positive (and negative) sides of the hyperplanes are indicated by the full-dimensional cell whose signs [+++++] are all positive. Topes are represented in the upper left figure. Cocircuits are represented in the upper right figure. Other non-null covectors are represented in the third figure. This oriented matroid is obtained from the equations of the running example (Sec. 12.2.2).

Minors. Given a subset A of E, we define the oriented matroid $M\backslash A$ obtained by *deletion of A from M* as the oriented matroid defined as above from the vectors in $E \setminus A$ (its covectors can be characterized in a combinatorial way, but we omit this). Observe that a representation of M directly yields a representation of $M\backslash A$. An example of deletion is shown in the left side of Fig. 12.5.

Given a subset A of E, we define the oriented matroid M/A obtained by *contraction of A from M* in the following way (again a combinatorial characterization exist that we omit). Geometrically, the contraction of A

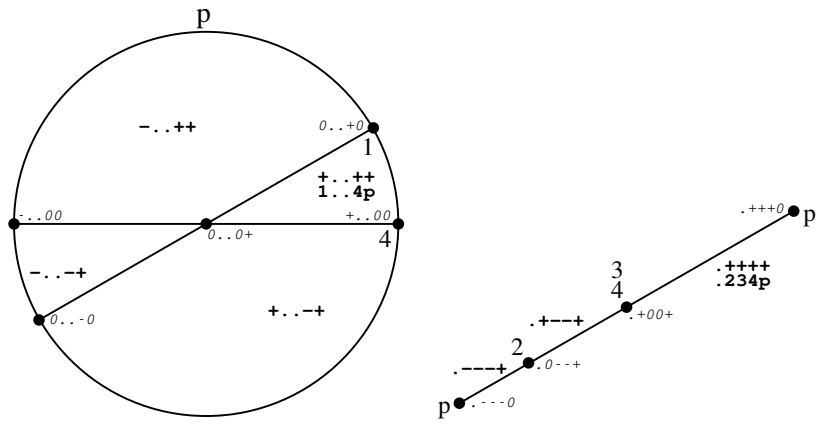


Fig. 12.5 Oriented matroid minors. Spherical representations of the oriented matroid $M\backslash 23$ on the left and of the oriented matroid $M/1$ on the right, where M is the oriented matroid of Fig. 12.4. Their topes and cocircuits are indicated (dots stand for omitted elements).

consists in considering only the set of cells contained in every hyperplane in A. Precisely, we consider the subspace $H_A = \bigcap_{a\in A} H_a$ as a new ambient space, and we consider the arrangement of hyperplanes $\{H_e \cap H_A \mid e \in E \setminus A,\ H_e \not\supseteq H_A\}$ indexed by $E \setminus A$ in this space (the elements e such that $H_e \supseteq H_A$ become so-called loops, which encode the null vector and are omitted in the structure). In this way, a representation of M directly contains a representation of M/A. An example of contraction is shown in the right side of Fig. 12.5.

The oriented matroids of type $M\backslash A/A'$ for $A \subseteq E$ and $A' \subseteq E$ with $A \cap A' = \emptyset$ are called *minors* of M. The ordering of the deletion/contraction operations does not matter since $M\backslash A/A' = M/A'\backslash A$.

As noted above, a representation of M directly yields a representation of $M\backslash A$. However, the dimension of the space spanned by $E \setminus A$ is not necessarily the same as the initial ambient space spanned by E, which was our assumption (that is, the rank of $M\backslash A$ can be smaller than the rank of M). In this case, removing elements from a spherical representation of M does not yield a proper spherical representation of $M\backslash A$ under the same assumption, and points in the resulting representation do not correspond to cocircuits anymore. This subtlety will be important for us. The next lemma states that cocircuits of minors of M can be seen as cocircuits of M as soon as the assumption is preserved.

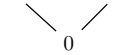
Lemma/Definition 12.2. Consider the minor $M' = M\backslash A/A'$ of M. Assume that the rank of $M\backslash A$ is equal to the rank of M. Then, for any cocircuit C' of M', there exists a unique cocircuit C of M such that $C' \setminus A = C$. We call C the *lift* of C' in M.

For example, consider the cocircuit [0..-0] of $M\backslash 23$ in Fig. 12.5. Its lift in M is the cocircuit [0---0] of M in Fig. 12.4.

12.3.2 *Encoding regions with super-covectors*

What follows does not belong to standard oriented matroid theory. We introduce original concepts for our application.

Let us define $\mathcal{Q} = \{\ 0,\ -,\ +,\ \times\ \}$ as the set of *super-signs*, where $\times$ is called *the undetermined sign*. The set $\mathcal{Q}$ is ordered with $0 \leq + \leq \times$ and $0 \leq - \leq \times$, forming a lattice (depicted below), whose *join* and *meet* operators are denoted by $\vee$ and $\wedge$, respectively:



Definition 12.3. Let E be a finite set. A *super-signed-set* over E (*sss* for brevity), is an element $U \in \mathcal{Q}^E$. We naturally extend, componentwise, the lattice structure of $\mathcal{Q}$ to a lattice structure of $\mathcal{Q}^E$. Let $U, V \in \mathcal{Q}^E$. We say that U is *smaller* than V, noted $U \preceq V$, if $U_e \leq V_e$, for all $e \in E$. Finally, $U \vee V$ and $U \wedge V$, are defined by $(U \vee V)_e = U_e \vee V_e$ and $(U \wedge V)_e = U_e \wedge V_e$, respectively, for all $e \in E$.

For $e \in E$, the super-sign associated to e in U is denoted by U_e. For a super-sign $\sigma \in \mathcal{Q}$, we define the set $U^\sigma = \{e \in E : U_e = \sigma\}$. Observe that if $U^\times = \emptyset$, then U is a signed-set, as addressed in Sec. 12.3.1.

Observe that two signed-sets U and V are conformal (Sec. 12.3.1) if and only if no element of $U \vee V$ has the undetermined sign. In that case, we have $U \vee V = U \circ V$. For example, we have [+0] $\vee$ [-0] = [x0], [+0] $\vee$ [0-] = [+0] $\circ$ [0-] = [+-], [+x] $\vee$ [00] = [+x], and [+x] $\vee$ [x0] = [xx]. Observe also that, for two signed-sets U and V, if $U \preceq V$ then U and V are conformal.

Let M be an oriented matroid on E as built in Sec. 12.3.1. Recall that each $e \in E$ is associated with one equation that partition the space into one hyperplane H_e^0 and two (open) halfspaces H_e^+ and H_e^-.

Property 12.1. For a sss U over E, the following conditions are equivalent:

- the following region is non-empty:
$$\text{region}(U) = \bigcap_{\substack{e\in E\\ U_e\neq\times}} H_e^{U_e}$$
- the signed-set $U\setminus\left(U^{\times}\cup U^{0}\right)$ on $E\setminus\left(U^{\times}\cup U^{0}\right)$ is a maximal covector (or tope) of the minor $M\backslash U^{\times}/U^{0}$ of M.

Definition 12.4. A sss U satisfying the conditions of Property 12.1 is called a *super-covector* of M, or *scovector* for brevity. Also, region(U) is called the *region of* U; and $\overline{\text{region}}(U)$ denotes the topological closure of region(U). Two super-covectors U and V representing the same region are considered as *equivalent* and noted $U \sim V$. The family of all scovectors of M will be denoted by $\mathcal{SC}\text{ov}$.

We call *null-scovector* the scovector with only `0` signs. Note that its region is $\{0\}$, consisting of the null-vector only. Note also that the scovector with only $\times$ signs yields the whole ambient space as a region.

12.3.3 *Super-covectors versus covectors*

Super-covectors essentially consist of groups of covectors/regions induced by covectors/regions of minors. Obviously, covectors of M are also scovectors. If C is a covector, then region(C) is the cell of C as addressed in the previous section. The scovectors with no 0 sign and no $\times$ sign are the conventional topes (or maximal covectors) of the oriented matroid (Sec. 12.3.1). The scovectors with no 0 sign are called *super-topes* of M. We mention that this notion was addressed under the same name in [23] and is equivalent to the so-called notion of T-convex sets; see [21, Sec. 4.2].

As an example, Fig. 12.6 shows various scovectors of the oriented matroid of Fig. 12.4. Observe how different scovectors may represent the same region. For instance, in Fig. 12.6: the scovectors [`+++0+`], [`+xx0+`] and [`xx+0+`] represent the same region, which is the cell of the covector [`+++0+`] of M (and the cell of the covector [`+++.+`] of the minor $M/4$); the scovectors [`0x--+`], [`0x-x+`] and [`0xx-+`] represent the same region, which is also the cell of the covector [`0.--+`] of the minor $M\backslash 2$ and the cell of the covector [`..--+`] of the minor $M/1\backslash 2$.

Definition 12.5. For two scovectors U and V, let us denote $U \trianglelefteq V$ if, for all $e \in E$, we have $U_e = V_e$ or $V_e = \times$. Equivalently: $U \trianglelefteq V$ if and only if $U \preceq V$ and $U^0 \subseteq V^0 \cup V^{\times}$.

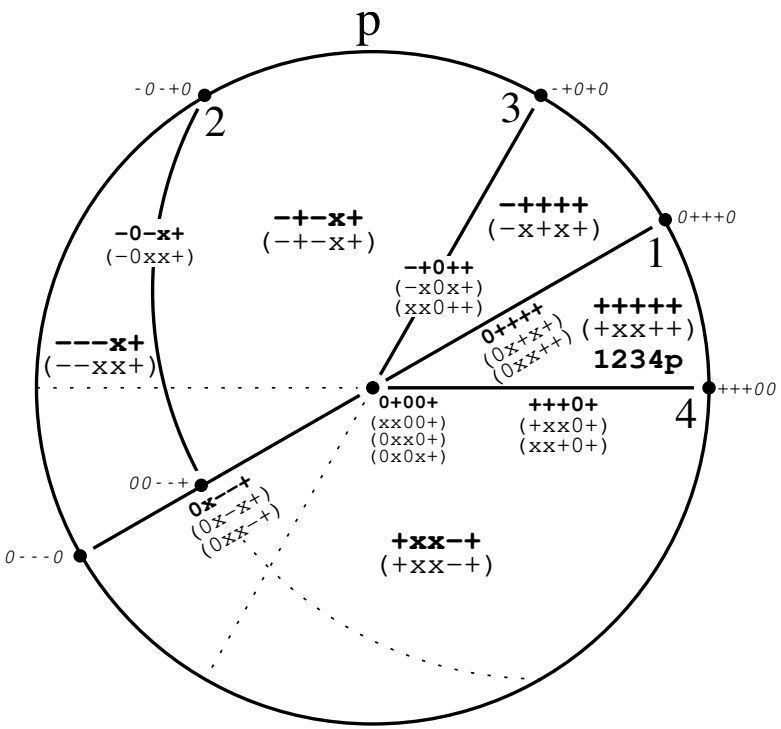


Fig. 12.6 The figure shows various super-covectors (or scovectors) for the oriented matroid of Fig. 12.4. Each scovector is written in its region (Definition 12.4). The bold scovector is the representative of the region (Proposition/Definition 12.7), and some equivalent scovectors are written below. The dotted portions of elements represent those which are signed $\times$ for the regions that they cross. The cocircuits, that will serve as borders of scovectors, are written in italics (Sec. 12.3.6). This set of scovectors forms a fragmentation of the oriented matroid, that is, their regions form a partition of the sphere (Definition 12.6). This fragmentation is obtained from the toy example addressed in Sec. 12.2.2 and Fig. 12.2.

As seen below in Property 12.3, this $\trianglelefteq$ relation corresponds to an inclusion relation for regions of scovectors. Observe that two covectors are not comparable for $\trianglelefteq$ unless they are equal. The $\trianglelefteq$ relation will be useful for handling scovectors representatives as defined in Sec. 12.3.4.

Property 12.2. For a scovector U, we have:

- covectors $C \trianglelefteq U$ are obtained from U by possibly replacing every $\times$ sign in U with a sign in $\{+, -, 0\}$, and keeping only covectors of M;

- covectors $C \preceq U$ are obtained from U by possibly replacing every $\times$ sign in U with a sign in $\{+,-,0\}$ and/or every $\{+,-\}$ sign in U with a 0 sign, and keeping only covectors of M;
- $\operatorname{region}(U) = \bigsqcup_{\substack{C \in \mathcal{C}\mathrm{ov} \\ C \trianglelefteq U}} \operatorname{region}(C)$ and $\overline{\operatorname{region}}(U) = \bigsqcup_{\substack{C \in \mathcal{C}\mathrm{ov} \\ C \trianglelefteq U}} \operatorname{region}(C)$ (where $\sqcup$ denotes a disjoint union).

For instance, in Fig. 12.6, the scovector [+xx−+] is the union of the following covectors from Fig. 12.4: [+−−−+], [++−−+], [+−+−+], [+++−+] (topes), [+0−−+], [+−0−+], [++0−+], [+0+−+] (intermediate covectors) and [+00−+] (cocircuit).

Definition 12.6. A *fragmentation* of M is a set $\mathcal{F}$ of scovectors of M whose regions partition the ambient space (that is, regions are disjoint to each other and their union equals the whole space). Equivalently, in a combinatorial way: the sets $\{C \mid C \in \mathcal{C}\mathrm{ov},\ C \trianglelefteq U\}$, for $U \in \mathcal{F}$, partition the set of covectors of M. (This is a combinatorial counterpart of Definition 12.1.)

12.3.4 *Representative of a super-covector*

As noticed above, various super-covectors may represent the same region. The notion of representative will be essential to compare and relate homodromies. Representative scovectors are written in bold in Fig. 12.6.

Proposition/Definition 12.7. Let U be a scovector of M. There exists a unique scovector of M, which we denote by $\operatorname{rep}(U)$, such that $U \sim \operatorname{rep}(U)$ and $\operatorname{rep}(U)^{\times}$ has the smallest possible number of $\times$ signs. It satisfies

$$\operatorname{rep}(U) = \bigwedge_{V \sim U} V = \bigvee_{\substack{C \in \mathcal{C}\mathrm{ov} \\ C \trianglelefteq U}} C.$$

If the region of U is $\{0\}$ (consisting of the null-vector only), then $\operatorname{rep}(U)$ is the null-scovector (with only 0 signs). We call *representative* of U, or of $\operatorname{region}(U)$, the scovector $\operatorname{rep}(U)$. A scovector which is equal to its representative is called a *representative scovector*. Clearly, covectors are representative scovectors.

Property 12.3. For two scovectors U and V, we have the following:

(i) $U \sim V$ if and only if $\text{rep}(U) = \text{rep}(V)$.
(ii) If $U \preceq V$ then $\text{rep}(U) \preceq \text{rep}(V)$, and if $U \trianglelefteq V$ then $\text{rep}(U) \trianglelefteq \text{rep}(V)$.
(iii) $\text{region}(U) \subseteq \text{region}(V)$ if and only if $\text{rep}(U) \trianglelefteq \text{rep}(V)$.
(iv) $\overline{\text{region}}(U) \subseteq \overline{\text{region}}(V)$ if and only if $\text{rep}(U) \preceq \text{rep}(V)$.

For instance, in Fig. 12.6, the scovectors [+++0+], [+xx0+] and [xx+0+] admit the same representative scovector [+++0+], and the scovectors [0x−−+], [0x−x+] and [0xx−+] admit the same representative scovector [0x−−+].

Note that "rep" cannot be omitted in the above statements. For instance, in Fig. 12.6, the two scovectors [x+00+] $\sim$ [0+x0+] yield the same region (with representative [0+00+]) but are not comparable.

Note also that, for two scovectors U and V, $\text{rep}(U \vee V)$ can be different from $\text{rep}(U) \vee \text{rep}(V)$. (Thus, in general, U is not equivalent to $\bigvee_{V \sim U} V$.) For instance, [0x−x+] $\vee$ [0xx−+] = [0xxx+] whose region is the whole part of H_1 in the positive side of H_p, larger than the region of [0x−−+].

However, with the above properties, one can see that the lattice structure of $\mathcal{Q}^E$ naturally induces a consistent lattice structure for the set of representative scovectors. (The join and meet operations between $\text{rep}(U)$ and $\text{rep}(V)$ are given by $\text{rep}(\text{rep}(U) \vee \text{rep}(V))$ and $\text{rep}(\text{rep}(U) \wedge \text{rep}(V))$, respectively.) This lattice is isomoprhic to a lattice for closures of scovector's regions.

Finally, representative scovectors can be considered as the useful scovectors for a practical encoding and handling of the regions. Practically, they can be computed using the construction of Sec. 12.3.6 below.

12.3.5 *Contiguity between super-covectors*

The notion of contiguity between homodromies will be central to use our model, and it can be directly deduced from the oriented matroid structure.

Property 12.4. Let U and V be two scovectors. Then $U \wedge V$ is a scovector.

We have $\overline{\text{region}}(U) \cap \overline{\text{region}}(V) \neq \{0\}$ (where 0 denotes the null-vector) if and only if the $\text{rep}(U \wedge V)$ is not the null scovector (with only 0 signs), that is, if and only if $\text{region}(U \wedge V) \neq \{0\}$.

Furthermore, if $\text{region}(U) \cap \text{region}(V) = \emptyset$ then the region of $U \wedge V$ is the greatest common face of the closures of the regions of U and V.

Definition 12.8. Two scovectors U and V with disjoint regions are called *contiguous* when the union of their regions is connected, that is, when one can pass continuously from $\text{region}(U)$ to $\text{region}(V)$.

Proposition 12.1. *For two scovectors U and V with disjoint regions, the following conditions are equivalent:*

(i) U *and* V *are contiguous;*
(ii) $\overline{\text{region}}(U) \cap \text{region}(V)$ *or* $\text{region}(U) \cap \overline{\text{region}}(V)$ *is non-empty;*
(iii) $\text{rep}(U \wedge V)$ *satisfies* $\text{rep}(U \wedge V) \trianglelefteq \text{rep}(U)$ *or* $\text{rep}(U \wedge V) \trianglelefteq \text{rep}(V)$.

Application. Further properties and underlying combinatorial structures can be obtained by exploiting, for instance, the approach of the above properties. Starting with the set of all representatives of scovectors of a fragmentation, and applying all possible $\wedge$ and $\vee$ operations, a lattice is naturally built. This lattice is a combinatorial representation of the way the algorithm acts on the input space and contains all the necessary information to perform qualitative reasoning and test data generation tasks. Such applications will be given in Sec. 12.4 using the above results as combinatorial criteria.

12.3.6 *Border of a super-covector*

Given a scovector U, our goal is here to compute $\text{rep}(U)$ by a join operation applied to a set of cocircuits of M, or of a suitable minor of M, that we call the *border* of U. This construction yields a way to compute representative scovectors, and, furthermore, to compute boundary test data in Sec. 12.4.3. Let us state a definition and result in the most general case, that we will explain and illustrate in the simpler most frequent cases. Various equivalent definitions are possible but will be omitted here.

Definition 12.9. Let U be a scovector of M. Let X be the union of all supports of cocircuits of M which are contained in $U^\times$. Let us denote $M' = M \backslash U^\times / U^0$ and $U' = U \setminus \left(U^\times \cup U^0\right)$. Consider the cocircuits of M' which are conformal to U'. Then, the *border of* U *in* M is the set of scovectors of M which are obtained by taking the lifts in $M \backslash X$ of these cocircuits of M', and then by adding a $\times$ sign to all elements in X. The border is denoted by $\partial_M(U)$, or by $\partial(U)$ for brevity.

Scholia. In Definition 12.9, in the case where $X = \emptyset$, which we call the *tame case*, $\partial(U)$ is a set of cocircuits of M. (In this case, equivalently, the rank of $M \backslash U^\times$ equals the rank of M, and it is the most frequent in applications.) In the general case, $\partial(U)$ is a set of very special scovectors

of M: for $V \in \partial(U)$, we have that $V^{\times} = X$ and $V \setminus X$ is a cocircuit of $M \setminus X$. Furthermore, $\partial_{M\setminus X}(U \setminus X) = \{V \setminus X \mid V \in \partial_M(U)\}$.

Theorem 12.2. *Let U be a scovector of M. Then* $\mathrm{rep}(U)$ *is the join of its border:*

$$\mathrm{rep}(U) = \bigvee_{C \in \partial(U)} C.$$

Furthermore, for two scovectors U and V, we have $U \sim V$ if and only if $\partial(U) = \partial(V)$.

First, let us consider the particular case where $U^{\times} = \emptyset$ (hence $X = \emptyset$) and $U^0 = \emptyset$. In this case, $U = U'$ is a maximal covector of $M = M'$, and $\partial(U)$ is simply the set of cocircuits of M which are conformal to U. Then, the above result can be written $\mathrm{rep}(U) = U = \bigcirc_{C\in\partial(U)} C = \bigvee_{C\in\partial(U)} C$, and it is essentially a reformulation of Theorem 12.1, which is given here by the central equality. (Obviously, $U = \mathrm{rep}(U)$ as it has no $\times$ sign, and the conformal composition can be replaced with the join operation as the two operations coincide for signed-sets which are conformal to each other, as observed in Sec. 12.3.2.) As mentioned in Sec. 12.3.1, the geometric interpretation of the above result is the following classical result in geometry: a bounded convex polytope (encoded by a covector) is the convex hull (encoded by the composition operation) of its extremal points (encoded by conformal cocircuits). In the case where $U^{\times} = \emptyset$ and, possibly, $U^0 \neq \emptyset$, we have exactly the same result and interpretation but in the minor $M' = M/U^0$, and the border of U is also formed by cocircuits of M.

For instance, in Fig. 12.6, for the maximal covector $U = [\texttt{+++++}]$, the border $\partial(U)$ is formed by $[\texttt{0+00+}]$, $[\texttt{0+++0}]$ and $[\texttt{+++00}]$, whose join equals U; and for the covector $V = [\texttt{-+0++}]$, the border $\partial(V)$ is formed by $[\texttt{0+00+}]$ and $[\texttt{-+0+0}]$, whose join equals V.

Second, consider now the case where $X = \emptyset$ (but, possibly, $U^{\times} \neq \emptyset$). In this case, we call U a *tame* scovector, and the border of U is also formed by cocircuits of M. By Property 12.1, the signed-set $U' = U \setminus \left(U^{\times} \cup U^0\right)$ is a maximal covector of $M' = M\backslash U^{\times}/U^0$. Since $X = \emptyset$, then the rank of $M\backslash U^{\times}$ equals the rank of M (this is a lemma which we omit). Hence, the lift notion from Lemma/Definition 12.2 is well-defined in M'. Consider the border $\partial_{M'}(U')$ of U' in M' as defined in the case above. Then the cocircuits in $\partial_M(U)$ are the lifts in M of cocircuits of M' belonging to $\partial_{M'}(U')$.

As discussed in Sec. 12.3.1, being tame means that the region of U in the ambient space of M and the region of U' in the ambient space of M' coincide. According to our experiments on program testing, most (or all) scovectors are tame in applications. For instance, all scovectors depicted in Fig. 12.6 are tame. The border of the scovector $U = [\texttt{+xx-+}]$ is formed by the cocircuits [0---0], [0+00+] and [+++00] whose join equals U. The border of the scovector [-0-x+] is formed by the cocircuits [-0-+0] and [00--+].

Third, in the non-tame case, scovectors of the border are no more cocircuits of M. By properties of the rank function, we have that $r(M\backslash X) = r(M\backslash U^{\times}) = r(M) - r(X)$. Hence the lifting notion is well-defined in $M\backslash X$. By Property 12.1, the signed-set $U' = U \setminus \left(U^{\times} \cup U^{0}\right)$ is a maximal covector of $M' = M\backslash U^{\times}/U^{0}$. Consider the border $\partial_{M'}(U')$ of U' in M' as defined in the first case. Then, the scovectors in $\partial_M(U)$ are obtained from $\partial_{M'}(U')$ as said in Definition 12.9 by using Lemma/Definition 12.2 in $M\backslash X$.

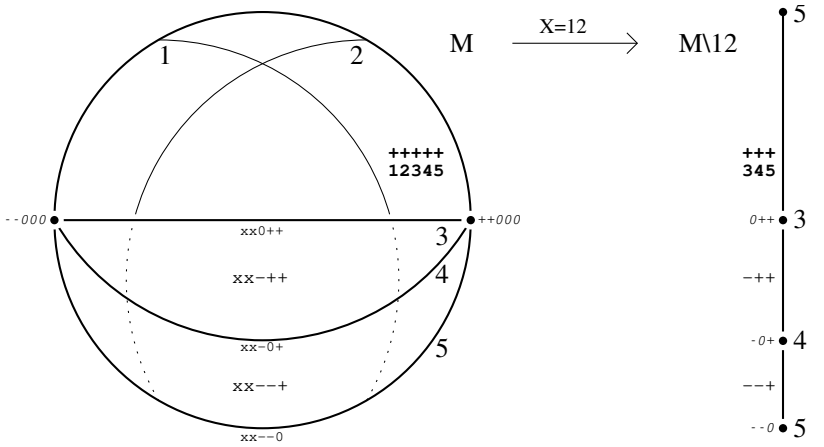


Fig. 12.7 Example of non-tame scovectors and their borders. The positive half-spaces are indicated by the region of the [+++++] covector. For any of the five depicted scovectors with $U^{\times} = 12$, we have that $U^{\times}$ contains the support $X = 12$ of the cocircuit [++000], hence these scovectors are non-tame. The spherical representation of $M\backslash X$ induced by the spherical representation of M in the sphere S^2 (on the left, with 5 at infinity) is thus not proper (the cocircuits do not correspond to points anymore, as shows the point associated with [++000] in M, which would be associated with [000] in $M\backslash X$ but which is not a cocircuit). The proper spherical representation of $M\backslash X$ is given in the sphere S^1 (on the right, with 5 still at infinity). The five above scovectors yield covectors of $M\backslash X$, as shown on its proper representation. The border of [..-++] in $M\backslash X$ is formed by [..0++] and [..-0+]. Thus, the border of [xx-++] in M is formed by [xx0++] and [xx-0+]. Observe on the left picture how the two corresponding half-linear-subspaces (of dimension 2 in $\mathbb{R}^3$) delimit the region of [xx-++].

An example of non-tame scovectors and their borders is given and explained in Fig. 12.7. Note that the scovectors belonging to a border are either cocircuits (tame case) or very special scovectors (cf. Scholia above). When they correspond to cocircuits of a minor $M\backslash X$ with a loss of rank, they are supported by half-linear-subspaces of the initial ambient vector space (which are just half-lines in the tame case).

Application. The above definition and result have their numeric counterpart that will be used for limit test data generation (see in Sec. 12.4.3). Each cocircuit of $\partial(U)$ can be realized as a real vector (recall that it corresponds a precise point in a spherical representation). In general, one can compute a real vector belonging to the region of each scovector of $\partial(U)$. Then, performing a join operation on the combinatorial structure amounts to perform a convex combination of the associated real vectors. In this way, by Theorem 12.2, one can generate real vectors on the extreme boundary faces — encoded by $\partial(U)$ — of the region of U, and in their neighbourhood inside the region of U.

12.4 How the model is used

Once the model (consisting of real inequalities and combinatorial information) has been computed (Sec. 12.2), one can determine the underlying oriented matroid which vehiculates additional combinatorial information such as representative scovectors and borders of homodromies (Sec. 12.3). We present in this section the three "outputs" of the qualitative model that one can derive from the previous computations: envision graphs, order-of-magnitude reasoning, and boundary test data generation. Our examples will still be based on our `WaterPhases` algorithm.

12.4.1 *Relations between Homodromies and the Envision Graph*

Intuitively speaking, an envision graph is a qualitative visualisation of the behaviour of the algorithm.

Homodromies form a partition of the ambient input space (called a *fragmentation* in Definitions 12.1 and 12.6). Moreover, homodromies, as regions of scovectors, are bijectively encoded in terms of their representative scovectors (Sec. 12.3.4). It is then natural to use the contiguity property of Proposition 12.1 as a straightforward combinatorial criterion for building the following adjacency graph.

Definition 12.10. Let us consider a *fragmentation* $\mathcal{F}$. The *homodromy graph* of $\mathcal{F}$ is the undirected graph G whose vertices are elements of $\mathcal{F}$ and whose edges are pairs of contiguous scovectors in $\mathcal{F}$.

In practical applications, homodromy graphs are relatively dense graphs, since they express the geometric configuration of all the feasible paths of the algorithm. It is then tempting to aggregate neighbor vertices with respect to a user-defined equivalence relation $\simeq$. For instance, distinct homodromies can correspond to the same output of the initial software. In our example, $\simeq$ will be "having the same output" (see Figs. 12.8 and 12.9). In other cases, the equivalence relation can be established using the subset of inputs that are numerically influencing outputs (see example below and Fig. 12.10). More formally:

Definition 12.11. Let G be the homodromy graph of the fragmentation $\mathcal{F}$, and let $\simeq$ be an equivalence relation over $\mathcal{F}$. The *envision graph* of $\mathcal{F}$ with respect to $\simeq$ is the graph obtained from G by contracting edges with equivalent endpoints. The resulting vertices of G' are consistently labelled with the equivalence classes of $\simeq$.

In Fig. 12.8 is given the homodromy graph of Fig. 12.6. Its corresponding envision graph (where homodromies having the same output are aggregated) is illustrated in Fig. 12.8 and refined in Fig. 12.9. As illustrated in this example, the envision graph can also be refined as a directed graph in order to take into account the influence of increasing/decreasing variable changes.

As another example, let us consider the following source code for the algorithm $\texttt{MAX}_N$ which computes the maximum of an array of N elements.

```
max = a[0]
for i in range(1, N):
   if a[i] > max: max = a[i]
return max
```

In Fig. 12.10, are given the envision graphs of the $\texttt{MAX}_3$ and $\texttt{MAX}_4$, for the equivalence relation $\simeq$ defined as follows: two homodromies are equivalent if their output is influenced by the same input variable. This information is automatically collected during the TDT construction (Sec. 12.2.3). More generally, the algorithm $\texttt{MAX}_N$ admits 3^{N-1} homodromies (feasible paths) and its underlying oriented matroid is a classical one known as the *braid arrangement* (of dimension $N-1$, admitting $N!$ full-dimensional regions, in

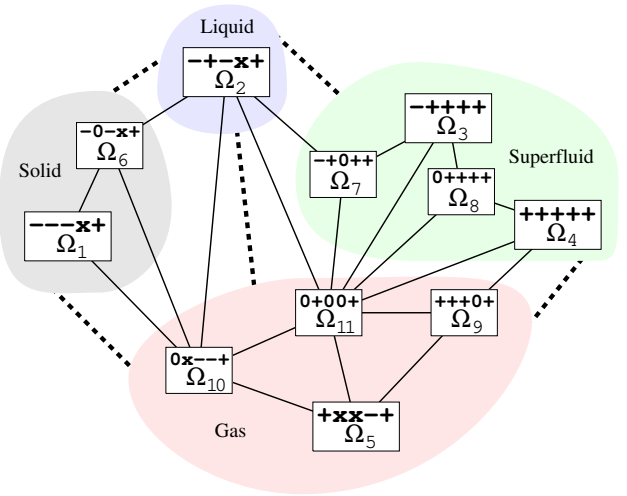


Fig. 12.8 Homodromy graph for the fragmentation of Fig. 12.6: vertices are the scovectors of Fig. 12.6, and edges reflect the contiguity relations between regions in Fig. 12.6. Envision graph for this fragmentation with respect to the output of the `WaterPhases` algorithm of Sec. 12.2.2 and Fig. 12.2: vertices are groups of scovectors corresponding to the same output, edges (represented by dashed segments) are induced by edges of the homodromy graph between those groups. Note that each homodromy is given with its representative scovector (Sec. 12.3.4), different from its initial scovector (Table 12.1).

bijection with permutations, or acyclic orientations of the complete graph). It contains $2^N - 2$ cocircuits. Among them, $2N - 1$ are sufficient to express combinatorially all the homodromies by means of their borders (Sec. 12.3.6). The envision graph of $\texttt{MAX}_N$, according to $\simeq$, turns out to be the complete graph K_N, which we refine in the following way: to any given input a_i corresponds a node labelled a_i (expressing the fact that the a_i input is influencing the output), admitting $N - 1$ outgoing arrows a_j, $j \neq i$, and $N - 1$ ingoing arrows, all equal to a_i.

Envision graphs are mainly destinated to the visualisation of local behaviours. When equations are linear, it is possible to have a global vision. However, when equations become very numerous, the envision graph can be very complex to visualize globally. It is then preferable to use it as a simulation tool (the simulation aspect is not in the scope of this paper).

Let us end with an application of the intersection criterion of Property 12.4. Suppose that one wishes to know what happens when three different phases meet at a common boundary surface. Using the four phases

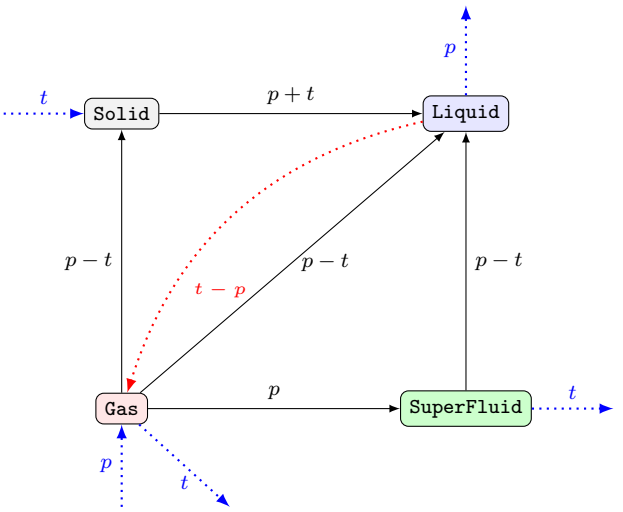


Fig. 12.9 Refined envision graph of the `WaterPhases` algorithm, automatically generated by our tool (with some minor cosmetic changes). The underlying graph is the same as the one given in Fig. 12.8. Labels p and t stand for *pressure* and *temperature*. The node `SuperFluid` aggregates the 4 contiguous `SuperFluid` homodromies Ω_3, Ω_4, Ω_7, Ω_8. The arrow from `Gas` to `Liquid` labeled $p-t$ expresses that *pressure* must *increase* and/or the temperature must decrease to allow the transition. Arrows are reversible: the previous arrow could be reversed from `Liquid` to `Gas` and labeled $t-p$ (dotted arrow). Arrows with no predecessors/successors (infinity arrows, dotted) mean that any increase/decrease will not change the phase. For instance, in the `Gas` phase, any increase of the *temperature* and any decrease of *pressure* will not change the phase; equally, only a *pressure* increase will allow the transition to `SuperFluid` phase. Note also that there is no possible direct transition from `Solid` to `SuperFluid`.

homodromies of maximal dimension, Ω_1(`Solid`), Ω_2(`Liquid`), Ω_3(`SuperFluid`), Ω_5(`Gas`) and this criterion, one directly gets the possible triple points:

- $\text{rep}(\Omega_1) \wedge \text{rep}(\Omega_2) \wedge \text{rep}(\Omega_3) = [\texttt{---x+}] \wedge [\texttt{-+-x+}] \wedge [\texttt{-++++}] = [\texttt{-00++}] \sim [\texttt{00000}]$ which corresponds to the null vector, which does not belong to the affine initial input space (outside the specifications). Hence, there is no possible triple point between the `Solid`, `Liquid` and `SuperFluid` phases.
- $\text{rep}(\Omega_1) \wedge \text{rep}(\Omega_2) \wedge \text{rep}(\Omega_5) = [\texttt{---x+}] \wedge [\texttt{-+-x+}] \wedge [\texttt{+xx-+}] = [\texttt{00--+}] = \text{rep}(\gamma_1)$
- $\text{rep}(\Omega_2) \wedge \text{rep}(\Omega_3) \wedge \text{rep}(\Omega_5) = [\texttt{-+-x+}] \wedge [\texttt{-++++}] \wedge [\texttt{+xx-+}] = [\texttt{0+00+}] = \text{rep}(\gamma_2)$

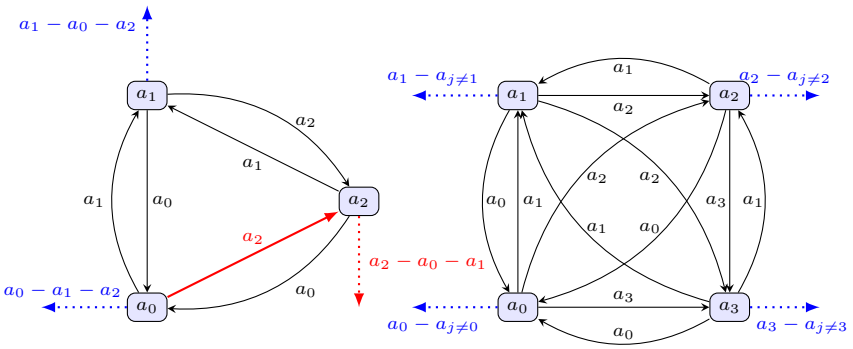


Fig. 12.10 The Envision Graphs of $\texttt{MAX}_3$ (left) and $\texttt{MAX}_4$ (right) algorithms, refined with arrows. Nodes are automatically labeled, according to the (here unique) input influencing the result. The arrow from a_0 to a_2, labeled a_2 means that when the maximum is the first element of the array (a_0), and the third element (a_2) is incremented then, eventually, a_2 will be selected as the maximum. If a_2 "continues" to increase or any other element decreases, the algorithm will stay at the same state a_2 (infinite arrow labeled $a_2 - a_0 - a_1$). Each of the 4 nodes of $\texttt{MAX}_4$ graph represents a 4-dimensional polytope. (One can note that reversing the arrows, yields the envision graphs of similar $\texttt{MIN}_3$ and $\texttt{MIN}_4$ algorithms.)

where γ_1 and γ_2 are indicated in Figs. 12.2 and 12.11 (we use representatives of homodromies, shown in Figs. 12.6 and 12.8, possibly different from their initial scovectors from Table 12.1).

12.4.2 *Order-of-Magnitude Reasoning and Qualitative Execution*

We introduce in this section an original order-of-magnitude algebra which can be considered as a non-standard version of an interval based order of magnitude algebra from [16]. We then illustrate how this formalism can be consistently combined with the combinatorial spatial properties of scovectors, to obtain useful qualitative conclusions.

Order-of-Magnitude Reasoning. Non-standard analysis handles hyperreal numbers, that is, infinitesimally small numbers (called *infinitesimals*) and infinitesimally big (called *infinite*) numbers, with an equivalence relation, noted $\approx$ meaning "infinitesimally close"; see [24,25]. In our model, ϵ stands for any strictly positive infinitesimal, d for any standard strictly positive finite (but not infinitesimal) real number, and ω stands for any positive infinite number (and their opposites are represented using a $-$ sign).

The qualitative value associated to an input of the algorithm, say p, will be denoted $[p]$ and will be a closed interval. For instance $[p] \approx [\epsilon, \omega]$ stands for a strictly positive input p with an unknown order-of-magnitude. The interval $[-\omega, \omega]$ expresses undeterminacy; $[-d, \epsilon]$ stands for any finite real (possibly negative) that is inferior that any standard positive real number. The four interval arithmetic operations $(\oplus, \otimes, \ominus, \oslash)$ constitute an interval extension of the hyperreal operations. Some examples: $[d, d] \oplus [\epsilon, \epsilon] \approx [d, d]$; $[d, d] \ominus [\omega, \omega] \approx [-\omega, -\omega]$, $[\epsilon, \epsilon] \otimes [\omega, \omega] \approx [\epsilon, \omega]$; $[-d, \epsilon] \oplus [0, d] \approx [-d, d]$; $[-d, -d] \otimes [-\epsilon, \epsilon] \approx [-\epsilon, \epsilon]$.

Qualitative execution. Suppose in the following that, in the toy algorithm, the temperature is a very small number, of unknown sign ($[t] \approx [-\epsilon, \epsilon]$) and the pressure has a standard positive value ($[p] \approx [d, d]$). Using linearity of $e_2 = 10t + p$ and the neutrality of $[d, d]$, it is then straightforward to evaluate the qualitative value of e_2. In fact, all coefficients, considered as standard reals, ($[d, d]$) can be dropped and we obtain: $[e_2] \approx [10 \cdot t + 1 \cdot p] \approx [10] \otimes [t] \oplus [1] \otimes [p] \approx [d, d] \otimes [t] \oplus [d, d] \otimes [p] \approx [t] \oplus [p] \approx [-\epsilon, \epsilon] \oplus [d, d] \approx [d, d]$. We conclude that the equation e_2 will be positive. For the other equations one gets:

- $[e_1] \approx [t] - [p] \approx [-\epsilon, \epsilon] \ominus [d, d] \approx [-d, -d]$ thus its sign is $-$
- $[e_3] \approx [t] - [p] - [d, d] \approx [-\epsilon, \epsilon] \ominus [d, d] \ominus [d, d] \approx [-d, -d]$, with sign $-$
- $[e_4] \approx [p] - [d, d] \approx [d, d] \ominus [d, d] \approx [-d, d]$ thus its sign is unknown, $\times$

Grouping all equations' signs, and adding a + sign at the end (as before for the equation at infinity) we obtain the scovector $U =$ [-+-x+] which corresponds to the `Liquid` homodromy Ω_2. *We conclude that, when temperature is very close to zero and pressure has a normal (non-negligible) positive value, the result will be the* `Liquid` *phase.*

Let's take another example: *what happens when the temperature is very high and pressure is normal (non-negligible) but unknown?* We have $[t] \approx [\omega, \omega]$ and $[p] \approx [-d, d]$. Evaluating, as previously, equations signs, we obtain the scovector $V =$ [+++x+]. Using now the intersection criterion of Property 12.4, one can observe that two `Gas` homodromies, $\text{rep}(\Omega_5) =$ [+xx-+] and $\text{rep}(\Omega_9) =$ [+++0+], and the `SuperFluid` homodromy, $\text{rep}(\Omega_4) =$ [+++++], combinatorially intersect the scovector V. *We conclude that when the temperature is very high and the pressure is unknown, the only compatible results will be the* `Gas` *and* `SuperFluid` phases.

12.4.3 *Boundary Test Data Generation*

Boundary test data generation is the quantitative side of the model. Its principle resides in the transformation of combinatorial operators $(\vee, \wedge)$ in real valued operations. Cocircuits constitute the combinatorial and numerical buildings blocks of the model. Rephrasing it, in software testing teminology: by composing a restricted set of test points, it is possible to generate, automatically, new limit test points on the borders of all the possible execution paths of the algorithm. Test data generation takes two steps.

- Step 1: Combinatorial generation of a specific region B.
- Step 2: Numerical generation of a point inside B, using rational valued operations.

For this sake, we extensively use the border notion of Sec. 12.3.6. Let us consider a scovector U and its border $\partial(U)$. To each cocircuit or scovector C in $\partial(U)$, can be arbitrarily associated, a real vector, named sample(C), realizing C in region(C). Such a real vector can be easily computed from the initial equations (if C is a cocircuit, then sample(C) can be any real vector spanning the half-line contained in the line $\bigcap_{e\in E, C(e)=0} H_e$, in the same side as the region $\bigcap_{e\in E, C(e)\neq 0} H_e^{C(e)}$). For instance, samples of the cocircuits of our example (Figs. 12.2 and 12.6) are given in Table 12.2.

Then, by Theorem 12.2, a real vector in region(U) realizing $U = \bigvee_{C\in\partial(U)} C$ can be any combination of the real vectors sample(C) for $C \in \partial(U)$ with positive coefficients: sample(U) $= \sum_{C\in\partial(U)} \alpha_C$.sample($C$), with $\alpha_C \in \mathbb{R}^{*+}$ for all $C \in \partial(U)$. Furthermore, real vectors in the region in

Table 12.2 The 7 basic cocircuits of the toy model and the coordinates of their associated real samples. Cocircuit γ_1 = [00--+] means that, at the point $\gamma_1(0, 0, 1)$, e_1 and e_2 are zero, and e_3 and e_4 are negative. The fifth sign expresses its position at infinity (the positive sign means that the point is inside the specifications). The sample coordinates are given in the 3-dimensional vector space containing the 2-dimensional affine input space. The third coordinate equals 1 for points inside the specifications and 0 for the points "at infinity" (see General setting in Sec. 12.3.1). The 7 samples are represented in Figs. 12.2 and 12.11.

cocircuit	coordinates	cocircuit	coordinates
γ_1 [00--+]	[0, 0, 1]	γ_2 [0+00+]	[600, 30, 1]
c_1 [0---0]	[-1000, -50, 0]	c_2 [-0-+0]	[-100, 1000, 0]
c_3 [+++00]	[0.1, 0, 0]	c_4 [0+++0]	[2, 0.1, 0]
c_5 [-+0+0]	[1, 0.1, 0]		

the neighborhood of these boundary vectors can be generated (for instance) as sample$(C) + \varepsilon$.sample(U) for $C \in \partial(U)$ and for a small $\varepsilon \in \mathbb{R}^{*+}$. (In the case of an initial affine space, one has to divide all coordinates by the coordinate corresponding to the element at infinity, in order to get points inside the specifications.)

In this way, one can compute real vectors at all boundaries between homodromies as well as at the neighborhood of all boundaries in all homodromies that they bound. Such a set of real vectors, serving as a relevant limit test data, is illustrated with grey dots in Fig. 12.11, obtained from the fragmentation given in Fig. 12.6 for the toy example. Real coordinates of test data obtained in the above way are given in Table 12.1.

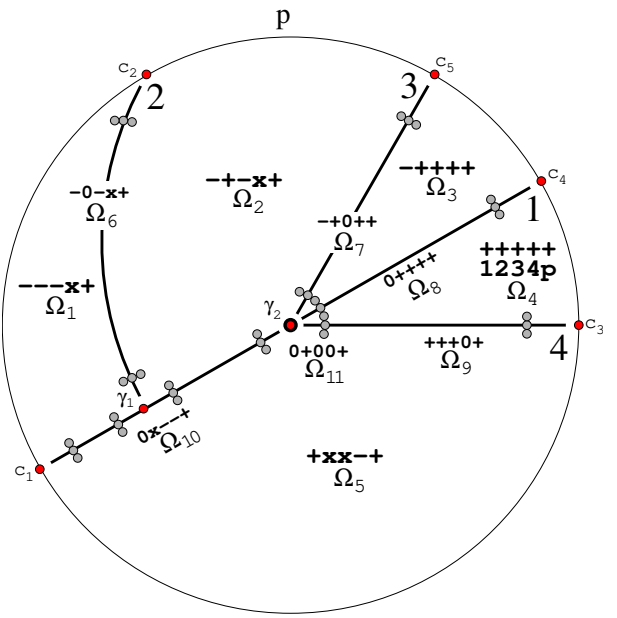


Fig. 12.11 Using scovectors and their borders to generate real vectors in homodromies of a fragmentation, in their boundaries, and in the neighborhood of their boundaries (Sec. 12.3.6), for the toy example of Sec. 12.2 and Fig. 12.2. The homodromies $\Omega_1, \ldots, \Omega_{11}$ correspond to the fragmentation of Fig. 12.6 and Table 12.1. The border cocircuits are represented by points denoted γ_1, γ_2, c_1, c_2, c_3, c_4, c_5, consistently with Fig. 12.2 and Table 12.2. Light grey dots represent samples (or test points) in homodromies, their boundaries, and their neighborhood; they are obtained by barycentric combinations of cocircuit points.

We now present some applications of the previous general construction to the computation of limit test points. The qualitative execution, in the previous section, permitted us to conclude that, *when temperature is very close to zero and pressure has a normal positive value, algorithm yields the* `Liquid` phase. Suppose now that this result contradicts the specifications which predict that, in that case, one should obtain the 'ice' result (the `Solid` phase). In other words, the analyzed algorithm contains a defect (bug).

The tester thus decides to examine what happens at temperatures close to zero in the boundary of the `Solid` homodromy Ω_1 and the `Liquid` homodromy Ω_2. In other words, she wishes to obtain boundary inputs that lye exactly on the Solidification/Liquefaction surface.

- Step 1: Starting from Table 12.1, $\Omega_1 = $ `[--xx+]` and $\Omega_2 = $ `[-+-x+]`, one can compute their borders ($\partial(\Omega_1) = \{c_1, \gamma_1, c_2\}$ and $\partial(\Omega_2) = \{\gamma_1, \gamma_2, c_4, c_2\}$) hence, their representatives, getting $\text{rep}(\Omega_1) = $ `[---x+]` and $\text{rep}(\Omega_2) = $ `[-+-x+]`. Using Property 12.4, one can calculate their common face with the meet $\wedge$ operator: $\text{rep}(\Omega_1) \wedge \text{rep}(\Omega_2) = $ `[---x+]` $\wedge$ `[-+-x+]` $= $ `[-0-x+]` $= \text{rep}(\Omega_6)$. In other words, the homodromy Ω_6 (the open segment $]\gamma_1, c_2[$), forms the solidification/liquefaction separating surface between the two phases.
- Step 2: Since the tester wishes to examine what happens with a small temperature and a positive pressure, she may ask for a test data that is close to $\gamma_1(0,0)$, still staying inside Ω_6. Taking as coefficients (for the convex hull combination) two strictly positive rationals λ_1 and λ_2 with $\lambda_1 + \lambda_2 = 1$, with, say, $\lambda_1 = 9999 \times 10^{-4}$, "much closer" to 1, than $\lambda_2 = 10^{-4}$. The join $\vee$ operation is now translated into a convex sum of rational coordinates. We obtain a boundary test data, $T_1 = \lambda_1 \cdot \text{sample}(\gamma_1) + \lambda_2 \cdot \text{sample}(c_2) = 9999 \cdot 10^{-4}$`[0,0,1]`$+10^{-4}$`[-100,1000,0]` $=$ `[-0.01,0.1,0.9999]`. Transforming the last coordinates in affine coordinates (dividing by the non-zero infinite third coordinate) one gets the test data $T_1 = $ `[-0.010001, 0.10001]`. One can check that T_1 nullifies the equation e_2, and still lies inside the `Solid` homodromy Ω_6. Exchanging λ_1 with λ_2 would still yield a point of Ω_6 but close to the specification frame.

What happens if one wishes to have an equivalent point of T_1 (close to γ_1) but in the `Liquid` side? Like previously:

- Step 1: The `Liquid` homodromy $\Omega_2 = [\texttt{-+-x+}]$ admits 4 border cocircuits, $\{\gamma_1, \gamma_2, c_4, c_2\}$, with $\text{rep}(\Omega_2) = [\texttt{-+x+}] = \gamma_1 \vee \gamma_2 \vee c_4 \vee c_2$.
- Step 2: For producing a limit test data point, T_2 close to $\gamma_1(0,0)$, but staying inside the `Liquid` homodromy we choose four positive rationals $\lambda_{1..4}$ such that their sum equals 1. Choosing say $\lambda_1 = 997 \cdot 10^{-5}$ and $\lambda_2 = \lambda_3 = \lambda_4 = 10^{-5}$, we obtain: $T_2 = \lambda_1 \cdot \text{sample}(\gamma_1) + \lambda_2 \cdot \text{sample}(\gamma_2) + \lambda_3 \cdot \text{sample}(c_4) + \lambda_4 \cdot \text{sample}(c_2) = 997 \cdot 10^{-5}\texttt{[0,0,1]} + 10^{-5}\texttt{[600,30,1]} + 10^{-5}\texttt{[2,0.1,0]} + 10^{-5}\texttt{[-100,1000,0]} = \texttt{[0.00502, 0.001301, 0.00998]}$. In affine coordinates one gets the test data $T_2 = \texttt{[0.503006, 0.130361]}$, yielding a `Liquid` output.

12.5 Conclusion

We presented an algorithmic qualitative model intended to assist the tester during the unit testing process. This model permits the visualization, in an abstract level, of the algorithmic behaviour. The spatial properties are based on oriented matroids. The propagation of orders-of-magnitude enables the validation of abstract properties concerning boundary behaviour. The combinatorial properties of cocircuits are finally used to generate concrete numeric test sets on any critical surface of any dimension.

Our approach is a technique allowing the automatic generation of test data. As we stressed in the introduction, this constitutes a critical open issue in the software engineering process, thus many other techniques have been proposed to tackle this problem. Note however that our testing data generation approach is not based on a structural coverage objective (i.e. cover all the statements of the source code), nor model based (i.e. use of an oriented graph), nor heuristic (like evolutionary techniques). It produces, formally and exhaustively (under certain assumptions and thanks to the mathematical theory presented in the previous sections), all the feasible execution paths of the source code. It is thus stronger (in the sense of [26]) than any other structural (path, branch, du-path, etc.) coverage objective.

This work is still at a prototype stage and is intended for software units whose conditions depend on decimal or rational inputs. Note however, that this restriction does not concern outputs which can be of any arbitrary type, as long as one disposes of an equivalence relation to aggregate them.

As a first experiment to evaluate its usefulness in the development process, the prototype was used in a first year Python programming course. The pedagogical objective was twofold: first, sensitize the students to the testing process by making them compare their (manually produced) test data with the automatic ones. Second, thanks to the envision graph, it gives to the students a more abstract and visual view of what was effectively coded, that can be compared with their qualitative understanding of the initial specifications.

A limitation obviously concerns the number of equations that the tool can analyze (limited in our prototype to 50), a number which can become very large in the case of large arrays and/or nested iterations. Note however, that the tool disposes of several filtering (slicing) options that can reduce drastically the number of studied paths.

We are currently working on several technical and theoretical future developments: the construction of a qualitative formal proof engine that checks properties by examining all possible paths (or in certain directions) of the homodromy graph; a qualitative simulation engine (that could be compared to a qualitative debugger) coupled with a decorated source code browser, which visualizes, step by step, the path followed on the homodromy graph according to qualitative input directions or magnitudes; and further useful mathematical properties in terms of oriented matroids.

Acknowledgments

This research was supported by: the OMSMO Project (Oriented Matroids for Shape Modeling) - Grant "Chercheur d'avenir 2015" (Région Occitanie & Fonds Européen de Développement Régional FEDER); the ANR Grant DISTANCIA (Metric Graph Theory, ANR-17-CE40-0015); and the ARCHIMEDES III (Greek minister of Research) project (support for TEI Larissa, No. 16, 2010: Application of Genetic algorithms and Qualitative Reasoning for intelligent software testing).

References

[1] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas and K. Karapoulios, Application of genetic algorithms to software testing (application des algorithmes génétiques au test des logiciels), in *5th International Conference on Software Engineering and its Applications, Toulouse, France*, pp. 625–636 (1992).

[2] M. Schoenauer and S. Xanthakis, Constrained GA optimization, in *Proceedings of the 5th International Conference on Genetic Algorithms, Urbana-Champaign, IL, USA*, pp. 573–580 (1993).

[3] R. P. Pargas, M. J. Harrold and R. R. Peck, Test data generation using genetic algorithms, *Software Testing, Verification and Reliability* **9**, pp. 263–282 (1999).

[4] P. McMinn, Search-based software test data generation: A survey, *Software Testing, Verification and Reliability* **14**, 2, pp. 105–156 (2004).

[5] O. Bühler and J. Wegener, Evolutionary functional testing, *Computers and Operations Research* **35**, pp. 3144–3160 (2008).

[6] S. Di Alesio, L. Briand, S. Nejati and A. Gotlieb, Combining genetic algorithms and constraint programming to support stress testing of task deadlines, *ACM Transactions on Software Engineering and Methodology* **25**, pp. 1–37 (2015).

[7] C. Sharma, S. Sabharwal and R. Sibal, A survey on software testing techniques using genetic algorithm, *International Journal of Computer Science Issues* **10** (2013).

[8] Z. Zhu and L. Jiao, Improving search-based software testing by constraint-based genetic operators, in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO 19. Association for Computing Machinery, New York, NY, USA, pp. 1435–1442 (2019).

[9] S. Xanthakis, S. Karapoulios, R. Pajot and A. Rozz, Immune system and fault tolerant computing, in *Artificial Evolution*, Vol. 1063. Lecture Notes in Computer Science, Springer-Verlag, pp. 181–197 (1996).

[10] Y. Jia and M. Harman, Constructing subtle faults using higher order mutation testing, in *8th International Working Conference on Source Code Analysis and Manipulation (SCAM 2008). Beijing, China, IEEE Computer Society* (2008).

[11] L. C. Briand, J. Feng and Y. Labiche, Using genetic algorithms and coupling measures to devise optimal integration test orders, in *14th IEEE Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy*, pp. 43–50 (2002).

[12] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer and R. S. Roos, Time aware test suite prioritization, in *International Symposium on Software Testing and Analysis (ISSTA 06). Portland, Maine, USA, ACM Press*, pp. 1–12 (2006).

[13] S. Yoo and M. Harman, Pareto efficient multi-objective test case selection, in *International Symposium on Software Testing and Analysis (ISSTA 07), ACM Press*, pp. 140–150 (2007).

[14] D. S. Weld and J. de Kleer, *Readings in Qualitative Reasoning about Physical Systems*. Morgan-Kaufman (1990).

[15] D. S. Weld and J. de Kleer, in D. S. Weld and J. de Kleer (eds.), *Qualitative Reasoning*. Series in Artificial Intelligence (1989).

[16] L. Travé-Massuyès and N. Piera, The orders of magnitude models as qualitative algebras, in *IJCAI'89: Proceedings of the 11th international joint conference on Artificial intelligence - Volume 2*, pp. 1261–1266 (1989).

[17] D. S. Weld, Exaggeration, in D. S. Weld and J. de Kleer (eds.), *Readings in Qualitative Reasoning About Physical Systems*. Morgan Kaufmann, pp. 417–421 (1990).
[18] S. Parsons and M. Dohnal, The qualitative and semiqualitative analysis of environmental problems, *Environmental Software* **10**, pp. 75–85 (1995).
[19] R. Moratz, *Qualitative Spatial Reasoning*. Encyclopedia of GIS, Editors: Shashi Shekhar, Hui Xiong, Xun Zhou (2017).
[20] K. D. Forbus, *Qualitative Representations How People Reason and Learn about the Continuous World*. MIT Press (2019).
[21] A. Björner, M. Las Vergnas, B. Sturmfels, N. White and G. Ziegler, *Oriented Matroids*, *Encyclopedia of Mathematics and Its Applications*, Vol. 46, 2nd edn. Cambridge University Press (1999).
[22] S. Xanthakis and E. Gioan, A qualitative reasoning model for software testing, based on oriented matroid theory, Full length journal paper (detailing and completing the present chapter) (In preparation).
[23] J. Edmonds and A. Mandel, *Topology of oriented matroids*. Ph.D. Thesis of A. Mandel, University of Waterloo (1982).
[24] G. Reeb, Analyse non standard (essai de vulgarisation), *Bulletin de l'APMEP* **32** (1981).
[25] R. Goldblatt, *Lectures on the Hyperreals. An introduction to nonstandard analysis*, Vol. 188. Graduate Texts in Mathematics. Springer-Verlag MR164 (1998).
[26] E. J. Weyuker, Comparing the effectiveness of testing techniques, in *Formal Methods and Testing*, Vol. 4949. Lecture Notes in Computer Science, Springer, pp. 271–291 (2008).

This page intentionally left blank

AI for Software Debugging

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0013

Chapter 13

AI-based Spreadsheet Debugging

Konstantin Schekotihin[a,*], Birgit Hofer[b,†], Franz Wotawa[b,‡] and Dietmar Jannach[a,§]

[a] *Alpen-Adria-University Klagenfurt, Austria,* [b] *TU Graz, Austria*
[*] *konstantin.schekotihin@aau.at,* [†] *bhofer@ist.tugraz.at,*
[‡] *wotawa@ist.tugraz.at,* [§] *dietmar.jannach@aau.at*

13.1 Introduction

Spreadsheets are by far the most popular end-user programming environment, and some estimate that there are about 1 billion users world-wide who work with spreadsheet environments like Microsoft Excel or Google Sheets.[1] The broad adoption of such end-user programming tools has two main reasons. First, the layout and structure of typical spreadsheets is in some ways similar to what one would probably end up with on paper. Second, users only need basic programming skills to develop their own spreadsheets. As a result, spreadsheets are used in all types of organizations and for various purposes such as storing information in a tabular form, budget estimations, and other kinds of computations that are required for decision-making. However, like any other software, also spreadsheets can contain faults, and there are plenty of known cases in which incorrect spreadsheets resulted in erroneous decisions and also huge financial losses.[2] [1] estimated that there is a chance of 3–5% for users to make a mistake when entering a formula in a spreadsheet. Consequently, a spreadsheet with 100 formulas has a probability of more than 95% to contain at least one fault.

[1] `https://www.grid.is/blog/excel-vs-google-sheets-usage-nature-and-numbers`.
[2] `http://www.eusprig.org/horror-stories.htm`.

Given this high probability for a complex spreadsheet to contain a fault, it is highly desirable to provide appropriate tools assisting spreadsheet developers in different aspects of quality assurance. A survey on spreadsheet quality assurance conducted by [2] shows that various methods exist to help users prevent, detect, localize, and repair faults in spreadsheets. For instance, to prevent faults researchers suggested numerous visualization frameworks, static spreadsheet analysis methods, as well as development methodologies such as model-oriented or test-driven development, or spreadsheet templates. Such fault prevention methods aim at the minimization of the risk for a fault to occur. However, various proposals were also made to enable users to *detect*, *localize*, and *repair* faulty parts of a spreadsheet.

Since modern spreadsheet environments only provide a limited and relatively simple set of tools for end-user developers, there is an increased interest, both in industry and academia, to enrich existing spreadsheet environments with better debugging support. In order to reduce the cognitive load of the users, many of the recent developments are focusing on automation of spreadsheet debugging.

In this chapter, we focus on modern approaches to spreadsheet debugging that are based on *Artificial Intelligence (AI) methods and techniques* to simplify the debugging process. Recent approaches like [3], for example, aim to automatically predict faults in spreadsheets with machine learning techniques, based on specific spreadsheet metrics and repositories of past faults. Other machine learning approaches like the Melford tool [4] utilize neural networks to detect faults in cells that erroneously contain numerical values instead of formulas. Training such machine learning approaches relies on repositories of known faults. An alternative AI-based approach uses automated reasoning that, differently from machine learning, does not require any historical training data. For instance, [5] use constraint satisfaction and model-based diagnosis to determine a set of formulae in a spreadsheet that, if considered faulty, would explain an unexpected calculation outcome.

Existing spreadsheet debugging approaches can be classified in various ways, e.g., by the type of information that they are using or the way they make a prediction that some part of a spreadsheet is faulty. The classification used in this chapter is designed by taking into account the fact that many of the approaches developed for automated debugging of general software artifacts were later also successfully applied to spreadsheet

debugging.[3] For instance, existing spreadsheet debugging approaches may consider structural information like dependencies between spreadsheet artifacts, but they may also use calculation traces to determine fault probabilities for certain units of a spreadsheet. An interesting aspect in that context is that spreadsheets can be much more restricted than software written with general-purpose programming languages, thus requiring less complicated debugging solutions. In spreadsheets, for example, the set of data types and operations is limited, user-defined functions are comparably rare, and there are no recursive calculations. All these properties make spreadsheets an interesting niche for the application of certain debugging methods and techniques.

In the following, we structure our review of AI-based spreadsheet debugging techniques in a way that is inspired by classification schemes used for automated software debugging methods see, e.g., [6]. Specifically, we classify existing approaches into two categories: *(i)* heuristic and statistics-based methods (Sec. 13.3), and *(ii)* model-based diagnosis techniques (Sec. 13.4). In the final section of the work, we give an outlook on possible future directions in this area. In particular, we point to potential limitations and open questions regarding the usability and the user interface design of advanced AI-based debugging tools. We believe that this aspect is particularly crucial for the success of such tools, because spreadsheet developers, i.e., the end-users, often do not have formal training or/and practical experience in software engineering.

13.2 Spreadsheet Debugging

The detection, localization, and repair of faults in spreadsheets can be tedious and error-prone since most of the spreadsheet processors, such as MS Excel or Numbers, do not provide sufficient support to their users. Consider the spreadsheet shown in Fig. 13.1a, which calculates the distance covered and the final speed of an object. When the user — usually a domain expert — inspects this spreadsheet, she might notice errors[4] in the computations. In particular, the accumulated distance, computed in cells B6, C6, and D6, should increase and the final velocity, calculated in cell E2, should be 0 instead of -20. In this case, to recognize these errors the end-user has to know the basic principles of Newtonian mechanics to understand

[3] For an overview of techniques and methods for fault localization in general software, see, e.g., [6].

[4] With the term error, we refer to unexpected calculation outcomes.

	A	B	C	D	E
1		**Acceleration**	**Constant Velocity**	**Deceleration**	**Final State**
2	**Initial Velocity [m/s]**	0	20	0	**-20**
3	**Acceleration [m/s²]**	2	0	-2	
4	**Duration [s]**	10	5	10	
5	**Distance [m]**	100	100	-100	
6	**Accumulated Distance [m]**	**100**	**200**	**100**	

(a) Value view

	A	B	C	D	E
1		**Acceleration**	**Constant Velocity**	**Deceleration**	**Final State**
2	**Initial Velocity [m/s]**	0	=B2+B3*B4	=B2+C3*C4	**=D2+D3*D4**
3	**Acceleration [m/s²]**	2	0	-2	
4	**Duration [s]**	10	5	10	
5	**Distance [m]**	=B2*B4+B3*B4*B4/2	=C2*C4+C3*C4*C4/2	=D2*D4+D3*D4*D4/2	
6	**Accumulated Distance [m]**	**=B5**	**=B5+C5**	**=B5+C5+D5**	

(b) Formula view

Fig. 13.1: Running example with a fault in cell D2.

that negative velocity, i.e., a backward motion of an object, is impossible with this amount of deceleration. Similarly, it should be clear that the notion of the accumulated distance means the total distance covered by an object and not its distance from the starting location. However, in many practical cases, a specification of a modeled problem might be unclear. As a result, the developers cannot spot a fault easily and must perform manual computations to verify the correctness of values computed by a spreadsheet processor.

The presence of errors in a spreadsheet indicates that there is at least one fault causing them. When the presence of faults is detected, the user has to locate their origins. A simple inspection of the values shown in Fig. 13.1a might provide an idea that the formula in D2 is likely to be faulty, because the object was already moving and therefore the initial velocity cannot be 0. Of course, this hypothesis must be verified, since a typo in some other formula might also lead to this unexpected result. In the formula view, shown in Fig. 13.1b, the user can see that the reference to the cell B2 is invalid. Whenever a fault is localized, there are multiple ways to repair it. One can modify the formula in D2 by replacing the reference to B2 with C2. Alternatively, one can extend the formula with an additional summand B3*B4. The identification of a correct repair gets considerably harder if a spreadsheet comprises multiple faults.

Examples of faulty spreadsheets, similar to the one given above, can be found in various document corpora, which were collected by researchers and practitioners over the last decade (see Sec. 13.5 for more details). These

corpora comprise spreadsheets from different sources and are widely used for the evaluation of novel spreadsheet debugging approaches. There are, however, some differences between the existing corpora. For instance, the spreadsheets in the *EUSES* and *Fuse* corpora were collected from the Internet and do not contain any ground-truth data indicating faults. Therefore, these corpora are mostly used in controlled studies, where faults are manually inspected by the researchers. Other corpora, like *Enron Errors*, *INFO1*, or *Hawaii Kooker*, were created by researchers using very specific sources, such as emails or practical assignments of students, and comprise annotations of identified faults. These corpora are particularly useful for the development and evaluation of automatic approaches, e.g., to train machine learning models helping end-users to find faults in spreadsheets. Further details about these approaches are provided in the next sections.

13.3 Heuristic and statistical approaches

The identification and localization of faults in software based on heuristics or statistical methods — and combinations thereof — has been pursued by many researchers over the last decades. Many ideas for developing such methods come from the general software engineering domain in which automated software debugging is a long-running research topic.

Heuristic methods for fault detection in general software artifacts were surveyed by [7]. Their study indicated that the prediction performance of the discussed approaches can depend on different factors, e.g., on the types of faults that occur in a particular piece of software or the underlying heuristic decision procedures. Architecture metrics, which focus on object-oriented structures, were in many cases better fault predictors than more traditional size and complexity metrics that measure, e.g., the complexity of statements or the numbers of methods.

Regarding *statistical* and machine learning techniques, [8] provides an overview of 90 papers that focus on the application of such techniques to fault prediction. The described methods use a variety of machine learning algorithms including decision trees, logistic regression, or deep neural networks. In many cases, however, the evaluation of such methods indicates that they can require substantial amounts of data for training the underlying models in order to achieve high precision and recall values in the fault identification process. The same observation may also hold in the domain of spreadsheet debugging.

Like in the debugging of general software artifacts, heuristic- and

statistic-based methods used in spreadsheets are based on the observation that developers often make similar mistakes. As a result, it is a reasonable approach in automated debugging to focus on the detection and localization of *typical* faults that are highly likely to occur in spreadsheets. Heuristic debugging approaches use a set of rules to detect patterns in a spreadsheet that correspond to typical faults. Quite often, if multiple heuristic rules are used, meta-heuristics are designed to decide if the results obtained by single heuristics are considered as an indication of a fault or not.

Statistical methods are in some ways similar to the heuristic ones, but use experience from previously debugged spreadsheets for the identification of faulty patterns in the data. Such methods usually first associate with every element of a spreadsheet that they consider, e.g., a formula or a cell, some probability that this element is faulty. Then, they often apply a decision procedure, e.g., a heuristic one, to decide if a spreadsheet element is considered faulty or not. In the following sections, we discuss heuristic- and statistic-based methods in more detail.

13.3.1 *Heuristic-based approaches*

Methods that use heuristic procedures are often based on the experience of their authors acquired while searching for bugs in spreadsheets. Like in the general software domain, however, the success of fault detection and localization procedures can depend strongly on the effectiveness and applicability of the underlying heuristics.

Early approaches to fault detection tried to copy the behavior of experienced spreadsheet programmers by identifying and checking which types of data are stored in cells [9, 10]. For instance, XeLda [11], UCheck [12], and Dimension [13] implement different approaches to data type inference in order to detect and localize faulty cells. Roughly speaking, these approaches first infer and assign data types to columns or rows of input cells, i.e., those cells that contain values and do not refer any other cells. Then, the assigned data types are recursively propagated over the formula cells, which are referring to cells that already have a known data type. UCheck derives the data types for input cells by analyzing the values of their column/row header cells, i.e, of the first cells positioned in the same row or column as the considered cells. Moreover, for each cell that contains a formula, UCheck uses the same procedure to derive the expected types of the formulas. The debugging algorithm reports a formula cell as possibly faulty if the type of this cell propagated from the input cells does not match the expected

type obtained from the header information. XeLda is designed similarly to UCheck but requires its users to manually annotate the data types for columns and rows. As a result, this tool works independently of the spreadsheet layout. Finally, Dimension reuses the header analysis of UCheck and extends it with a reverse type inference mechanism from formula to input cells and various data type conversions. Practical evaluations of UCheck and Dimension showed that in real-world scenarios these tools can identify almost 50% of all known faults in a given corpus of spreadsheets.

Example 13.1. In our example described in Sec. 13.2 types can be derived from the text of the first column and the first row. For instance, D2 can be labeled with "Initial Velocity" and "Deceleration". Its derived type consists of "Initial Velocity", "Acceleration", and "Duration". So the types are mismatched and D2 is declared as possibly faulty.

More recent tools such as AmCheck [14], as well as its successors CACheck [15] and EmptyCheck [16], extend the single-cell focus of the previously discussed approaches to arrays of cells. These tools are based on the assumption that formulas physically located near to each other in a spreadsheet are also logically related to each other. These approaches therefore use specific heuristics to extract arrays of formula cells that might contain faults. An array is defined as a set of columns or rows that comprises formula cells that are *(i)* functionally related and *(ii)* surrounded by "borders" of either empty cells or cells containing fixed values. Both AmCheck and CACheck predict a cell to be faulty whenever its formula deviates significantly from other formulas of the same array with respect to a set of predefined metrics. The EmptyCheck tool predicts if an empty cell in an array should contain a formula based on a clustering algorithm.

Example 13.2. Reconsider the example presented in Fig. 13.1. In addition to the types, array-oriented tools are also considering the nearby cells. For instance, C2 and D2 are considered as a part of an array, since they are surrounded by cells comprising fixed values and have similar types.

13.3.2 *Statistics-based approaches*

Designing a general heuristic procedure for fault prediction can be a tedious process. Statistic-based methods can simplify this problem by applying statistical or machine learning techniques to automatically identify faulty cells. One approach to automatically identify faults in spreadsheets is to

use *unsupervised learning* methods. The corresponding tools suggested in the literature, e.g., CUSTODES [17], WARDER [18], or SGUARD [19], first compute different features of spreadsheets and/or its parts and then use clustering algorithms to group similar cells with the goal of detecting outliers. These outliers are then considered to be the cells that are most likely a source of a fault. Technically, CUSTODES uses a two-stage technique utilizing strong and weak features of an analyzed spreadsheet. Strong features are extracted from groups of cells that are usually placed close to each other by spreadsheet processors, e.g., cells with formulas generated automatically by copying a formula over some range. Weak features are computed from cell labels, standard layouts, fonts, etc., that allow the tool to automatically adapt to varying tabulation styles.

Example 13.3. In our example, shown in Fig. 13.1, CUSTODES first converts all formulas into R1C1 notation that replaces absolute references to cells in formulas with relative ones. For instance, the formula in D2 is converted into '=RC[-1]+R[1]C[-2]*R[2]C[-1]', where R[2]C[-1] refers to a cell, which row number equals to the D2 row number plus two and column number to the D2 column number minus one. Next, the framework extracts an abstract syntax tree (AST) for each formula, e.g., D2 is translated into (+,RC[-1],*,R[1]C[-2],R[2]C[-1]), and generates a list of cell dependencies, e.g., (RC[-1],R[1]C[-2],R[2]C[-1]). These strong features are used in the first stage, where a clustering algorithm puts all cells with similar feature values in one cluster. Examples of such clusters include (B5,C5,D5) and (C2,D2). Next, for the cells of each cluster, the algorithm extracts the weak features listed above and tries to further refine clusters using weak features. This procedure, for instance, extends the cluster (C2,D2) to (B2,C2,D2) since B2 is structurally relevant to both C2 and D2. Finally, CUSTODES identifies that *(i)* B2 might be faulty since it contains a value, whereas C2 and D2 comprise formulas, and *(ii)* either C2 or D2 is faulty, since their formulas refer to different cells.

WARDER further refines the clustering technique used in CUSTODES by a more precise definition of weak features, e.g., occurrences of data cells among formula cells, cells whose content is different from the content of other cells in the cluster, etc. SGUARD implements a three-stage clustering method that *(i)* forms clusters of cells that perform similar computations and that are located in close proximity to each other; *(ii)* refines the clusters by removing irrelevant cells from clusters and unqualified clusters; and *(iii)* detects outliers based on the obtained clusters. Regarding

the evaluation of these clustering-based techniques, it was found that these tools were able to recognize up to 78% of the faulty cells in the *EUSES* [20] corpus.

ExceLint [21] is another tool that employs a custom clustering algorithm based on the reduction of the Shannon entropy. Like the tools listed above, it assumes that the structure of spreadsheets follows "rectangular-like" patterns, i.e., formulas in the same row or column are very likely to have the same semantics. Technically, the identification of the patterns in ExceLint is based on fingerprints, i.e., a two-dimensional vector representation that is generated for each formula. The value of each coordinate is defined by the sum of the relative column/row distances from the formula cell to other cells referred to in the formula. Given these fingerprints, a binary decomposition algorithm finds clusters in the spreadsheet by recursively dividing it into two parts such that each split minimizes the normalized Shannon entropy of the fingerprints in both subdivisions. Lastly, the tool generates repair suggestions for formulas where a deviation is identified. The suggestions are generated in a way that would lead to a moderate reduction of the overall entropy of the spreadsheet, which is typical for genuine fixes of faults.

Example 13.4. In our example, the formula in D2 is represented as a relative fingerprint (-4,-3), which is a sum of vectors corresponding to the referenced cells B2: (-2,0), C3: (-1,-1), and C4 : (-1,-2). The vector coordinates are computed by representing the spreadsheet as a Cartesian plane with the origin in D2. A range in a formula is represented by a vector with the same cardinality as the range. Given the fingerprint representation, the clustering algorithm identifies a cluster with C2 and D2, since their fingerprints are quite similar, but still have a small difference in the first coordinate.

On a set of 70 spreadsheets drawn from the *EUSES* corpus, the ExceLint tool was able to find around 63% of all formulas that were known to be faulty to the authors.

Another unsupervised method that combines both heuristic- and statistics-based methods was proposed by [22]. Their approach identifies faulty cells using *spreadsheet smells* [23–26], i.e., heuristic procedures that use spreadsheet metrics to identify possibly problematic parts of a spreadsheet. Technically, each smell can be viewed as a two-step procedure. First, smells are based on evaluating a certain metric on a given spreadsheet, where a metric is a function that maps some part of a spreadsheet to a

numeric value. Since each smell is designed to detect some specific quality issue, the underlying metric is defined to quantify to what extent the specific issue occurs in the input of a smell. Depending on the nature of an issue, a metric might take arrays of cells, single cells, or their content as input. In the second step, a heuristic that is specifically designed for each smell is used to decide if the value returned by the metric surpasses a certain threshold. If so, this indicates that the considered cell(s) might have some quality issues and should be inspected by the developers of the spreadsheet.

FaultySheet Detective [27] uses the output of spreadsheet smells to trigger a Spectrum-based Fault Localization (SFL) algorithm. Technically, this approach has two phases. In the first step, the tool computes two sets of cells: *(i)* the set of smelly cells and *(ii)* the set of output cells, i.e., those cells that refer to other cells, but themselves are not referred by any other cell. To find the smelly cells, FaultySheet Detective uses a catalog of 15 smells. These smells are, for example, designed to find formulas that point to empty cells or cells that exhibit suspicious patterns, like an empty cell appearing in a row that contains numerical values. Next, the algorithm determines calculation chains for all output cells and generates what is called a hit-spectra matrix in which every column corresponds to a cell in a spreadsheet and each row to an output cell.

Table 13.1: Hit-spectra Matrix Example.

c_1	c_2	...	c_n	*smelly*
1	0	...	1	0
0	1	...	0	1
...	...	...	...	...
0	0	...	1	0

The values 0 and 1 in the example matrix in Table 13.1 indicate: *(i)* if a cell c_i appears in the computation chain of an output cell, or *(ii)* if the computation chain represented by a row comprises smelly cells. In the second step, the tool uses the Ochiai coefficient [28], which is a popular suspiciousness measure widely used in SFL for fault localization in general software artifacts [29, 30]. FaultySheet Detective finally returns those cells to its user, where the value vectors are similar to the value vector of the *smelly* column in terms of the Ochiai coefficient, i.e., where the value of the coefficient surpasses some predefined threshold. An evaluation of the tool

on the Hawaii Kooker Corpus [31] of 73 faulty spreadsheets showed that given a proper similarity threshold and a relevant set of smells, the tool was able to detect up to 70% of the existing known faults.

To increase the accuracy of fault prediction, some spreadsheet debuggers use *supervised learning* techniques in case historical labeled data is available. Following such an approach, structures that are found in given sets of correct and faulty spreadsheets can for example be used to train machine learning models to identify incorrect placements of formulas and value cells in arrays. Melford [4] is an example of a method that implements such a machine learning approach based on Deep Neural Networks. It uses a training data set derived from a corpus of correct and incorrect spreadsheets to train a classifier that is able to automatically identify cells in which a value is erroneously placed instead of formula. To make such a prediction, Melford first converts a given spreadsheet into an image-like representation, where cells that contain values, formulas, and empty cells are mapped to different symbols. The obtained representation is then provided to a neural network that is trained to detect the cells that contain a value instead of a formula. An evaluation of the approach shows that Melford can successfully detect cells in which a number was erroneously placed instead of a formula.

Example 13.5. For the spreadsheet given in Fig. 13.1a, Melford generates the following mappings: A1 - O, A2 - S, B2 - N, D2 - F, denoting empty, string, number, and formula cell, resp. The converted spreadsheet is then provided to a trained classifier to identify those cells where the label must be F instead of S or N.

Another supervised approach, which uses smells for automatic fault detection and localization in spreadsheets, was recently proposed in [3]. The authors suggest to use machine learning methods and to train models able to predict if a cell in a given spreadsheet is faulty. The basis of their approach is an exhaustive catalog of spreadsheet metrics, which were specifically selected to detect different quality issues that might occur in spreadsheets. The catalog includes all metrics that were previously used in FaultySheet Detective, other metrics proposed in the literature e.g. [23–26], as well as new metrics designed in the context of the proposed method. Technically, given a set of faulty spreadsheets, in which every faulty cell is labeled as such, a training data set is generated using the following pipeline:

(1) The metrics from the catalog are applied to compute a value for every cell of each spreadsheet.
(2) The values are stored in a table in which every column corresponds to a metric and every row to a cell in a spreadsheet.
(3) All labels of the correct and faulty cells are also stored in the rows of the table.

Next, in the training phase, four machine learning algorithms — Deep Neural Networks [32], Support Vector Machines [33], Adaptive Boosting [34], and Random Forests [35] — were applied to train predictors of faulty cells. The results obtained for three corpora of faulty spreadsheets, *Enron Errors* [36], *INFO1* [37], and *EUSES* [38], indicate that the models trained by the Random Forest algorithm were able to correctly recognize 98%, 77%, and 61% of the faulty cells for each corpus, respectively. The Deep Neural Networks classifier exhibited good performance results on two of the real-world corpora. However, its performance degraded significantly for the *EUSES* corpus, which comprises spreadsheets with artificially injected (synthetic) faults.

13.3.3 *Empirical evaluations and limitations*

The experimental evaluations of heuristic- and statistic-based approaches, as described above, show that these methods demonstrate good accuracy and coverage of faults for which they were designed and/or trained. This success is based on the fact that the metrics used in the methods of both types are specifically designed using domain knowledge of experienced spreadsheet developers to discover quality issues. Given the high-quality information from these metrics, the decision-making procedures implemented as heuristics, correlation coefficients, or machine learning models are able to efficiently identify faults in real-world spreadsheets.

The main obstacles on the way to industrial applications of the discussed methods are their reliability and scalability. Both heuristic and statistical approaches are unable to discover new types of faults, e.g., those that were not considered by the developers of the metrics and those that were not present in the training data. In such application scenarios, the detection of novel types of faults may require significant changes in the decision-making procedures, e.g., re-training of machine learning models, and/or the creation of novel metrics that are able to discover such new types of faults. The creation of such metrics and the generation of relevant training data

sets can be costly since spreadsheets often contain sensitive information, which cannot be publicly shared. The problem might be partially solved by continuously extending the catalog of considered faults and by combining heuristic- or statistic-based approaches with model-based diagnosis techniques that we describe next.

13.4 Model-based diagnosis techniques

Already in the early years of AI research it was widely recognized that models of real-world artifacts can be used to identify and localize faults. The main idea behind Model-based Diagnosis (MBD) techniques is therefore to compare the observed behavior of an artifact with the behavior as predicted by a model. The artifact is considered as faulty if its observed behavior deviates from the predicted behavior. One of the first successful applications of MBD techniques was the localization of faults in digital circuits [39]. A model in this case encodes a circuit to be diagnosed, i.e., logic gates and circuit layout, as a set of logic sentences that represent the correct behavior of the circuit. Given the model and a set of circuit observations encoded as logical sentences, a diagnosis algorithm first infers if the observations are consistent with the model. Whenever inconsistencies are found, the diagnosis algorithm computes a set of gates that, if assumed to be faulty, explain the incorrect behavior of the circuit. Various MBD algorithms were proposed since the 1980s and, more importantly, MBD techniques were applied in various application domains, including software artifacts such as logic programs, knowledge bases, and spreadsheets. In the latter case, MBD techniques can support the user by analyzing which formulas could have caused unexpected outcomes and, vice versa, which formulas could *not* have caused the observed discrepancy.

In this section, we will first provide the basic definitions of MBD and show three types of models that can be used when applying MBD principles to spreadsheets. We will then discuss algorithms to compute the diagnoses and finally review existing tools and empirical results for MBD-based spreadsheet debugging.

13.4.1 *Basic definitions*

A classic MBD approach suggested by [39] defines a diagnosis problem as a triple (COMP, SD, OBS), which can be translated to the spreadsheet domain as follows:

- COMP is a set of identifiers for all formula cells, called *components*, e.g., D2 or E1;
- SD is a *system description*, which is a set of spreadsheet formulas, each represented as a logical sentence such as a constraint; and
- OBS is a set of *observations* represented by a set of logical sentences that encode values observed in the input and output cells of a spreadsheet.

According to [39], each component $C \in \text{COMP}$ is either abnormal, denoted $\text{AB}(C)$, or it behaves as expected. In this case an expected behavior of every component C, denoted $\mathit{Behav}(C)$, can be modeled in SD as follows $\neg\text{AB}(C) \rightarrow \mathit{Behav}(C)$. The latter logical sentence relates an assumption about the health state of a component with the requirement for this component to behave in a certain way. If we assume a component C to be faulty, i.e., $\text{AB}(C)$ is true, we correspondingly make no assumption about the behavior of C. Otherwise, $\text{AB}(C)$ is false and therefore the behavior of C represented by $\mathit{Behav}(C)$ is expected.

Ultimately, MBD aims at finding one or more subsets of COMP, called diagnoses, which might be responsible for the unexpected outcomes. Informally speaking, we try to find a set of components, which, if we assume them to be faulty, could in some ways explain the observed behavior. The following formal definition based on logical sentences does exactly that: for all components that are assumed to be faulty, the assumptions about their correct behavior are removed, and a diagnosis is found if this relaxed specification is consistent with the observed outcomes.

Definition 13.1 (Diagnosis). Given a diagnosis problem (COMP, SD, OBS). A subset $\Delta \subseteq \text{COMP}$ is a diagnosis, if and only if $\text{SD} \cup \text{OBS} \cup \{\neg\text{AB}(C) \mid C \in \text{COMP} \setminus \Delta\} \cup \{\text{AB}(C) \mid C \in \Delta\}$ is satisfiable. A diagnosis Δ is minimal if and only if no proper subset of Δ is a diagnosis.

Since spreadsheets mostly consist of arithmetic operations on numeric values, the encoding of the models and observations is usually done using constraint programming, which is then used to make the inferences that are necessary to compute the diagnoses. A Constraint Satisfaction Problem (CSP) is a tuple (V, D, CO), where V is the set of variables with the domains defined in the set D and CO is the set of constraints that connect the variables [40].

As mentioned before, in the spreadsheet domain, COMP corresponds to the identifiers of formula cells, SD models the formulas, and OBS encodes

one or more *test cases* that are defined by the user to reveal a fault. A test case $T = (I, O)$ for a spreadsheet is a pair of sets, where $I = \{(c, v)\}$ is a set of pairs of input cells and their values, and $O = \{(c, v_{exp})\}$ is a set of pairs of output cells and their *expected* values. Input cells are non-formula cells that are referenced by other cells; output cells are formula cells that are not referenced by other cells. The pairs in set I must specify values for all input cells which are required to compute values of all output cells listed in O. A test case *fails* if at least for one output cell the expected value differs from the computed one. Otherwise, a test case *passes*.

Example 13.1 (Test case). A test case for our running example (Fig. 13.1) is $I = \{(\text{B2}, 0), (\text{B3}, 2), (\text{B4}, 10), (\text{C3}, 0), (\text{C4}, 5), (\text{D3}, -2), (\text{D4}, 10)\}$ and $O = \{(\text{B6}, 100), (\text{C6}, 200), (\text{D6}, 300), (\text{E2}, 0)\}$. Since the computed values of D2, D6 and E2 differ from their expected values, this test case fails.

13.4.2 *Model types*

The system description SD for spreadsheet debugging can be modeled in three different abstraction levels:

- the *value-based model* [5, 41–43]
- the *dependency-based model* [44], and
- the *qualitative deviation model* [45].

The value-based model uses the values of the cells in the reasoning process. It computes a smaller number of cardinality-minimal diagnoses compared to the other models, but has the highest computation time. The dependency-based model uses correctness information regarding individual cell values in the reasoning process, i.e, a value is either correct or erroneous. This reduces the domain of all variables in V to Boolean. This model is less precise and usually returns more diagnoses than the value-based model. However, its computation time is lower. The qualitative deviation model lies in between the value-based and the dependency-based models. It uses additional information about the expected ranges for the cell values in the reasoning process, e.g., if a cell value should be larger, smaller, or equal to its current value. Therefore, the domains of all variables have exactly three possible values.

In detail, the formula cells of a spreadsheet $S = \{c_1, c_2, \ldots, c_n\}$ can be

translated into a **value-based constraint model $\mathbf{SD}_V(S)$** as follows:

$$\mathrm{SD}_V(S) = \bigcup_{i=\{1,\ldots,n\}} \left\{ AB_{index(c_i)} \vee c_i == constr(formula(c_i)) \right\}$$

where *index(c)* returns a unique index for each cell, the function *formula* returns the cell's formula, and where *constr* recursively translates a formula expression e as follows [43]:

- If e is a constant or a cell name c, then $constr(e) = c$.
- If e is of the form (e_1), then $constr(e) = (constr(e_1))$.
- If e is of the form $e_1 \circ e_2$, then $constr(e) = constr(e_1) \circ^* constr(e_2)$ where $\circ^*$ is the constraint representation of operator $\circ$.
- If e is $f(e_1, \ldots, e_n)$, then $constr(e) = f^*(constr(e_1), \ldots, constr(e_n))$ where f^* is the constraint representation of the function f.

Whenever an expression consists of sub-expressions, auxiliary variables tmp_i are introduced with i being a consecutive number to make the variables unique. Every test case $T = (I, O)$ is converted into value-based constraints as

$$OBS_V(T) = \bigcup_{(x,v)\in\{I\cup O\}} \{x == v\}.$$

Example 13.2 (Value-based model). The running example (Fig. 13.1) has the value-based system description $\mathrm{SD}_V = \{\mathrm{AB}_{\mathrm{B5}} \vee (\mathrm{B5} == \mathrm{tmp}_1 + \mathrm{tmp}_2), \mathrm{AB}_{\mathrm{B5}} \vee (\mathrm{tmp}_1 == \mathrm{B2} * \mathrm{B4}), \mathrm{AB}_{\mathrm{B5}} \vee (\mathrm{tmp}_2 == \mathrm{tmp}_3/2), \mathrm{AB}_{\mathrm{B5}} \vee (\mathrm{tmp}_3 == \mathrm{tmp}_4 * \mathrm{B4}), \mathrm{AB}_{\mathrm{B5}} \vee (\mathrm{tmp}_4 == \mathrm{B3} * \mathrm{B4}), \mathrm{AB}_{\mathrm{B6}} \vee (\mathrm{B6} == \mathrm{B5}), \ldots\}$. The test case from Example 13.1 translates to $\mathrm{OBS}_V = \{$B2==0, B3==2, ..., B6==100, C6==200, D6==300, E2==0$\}$.

For the **dependency-based model**, only the correctness information is propagated in the model:

$$\mathrm{SD}_D(S) = \bigcup_{i=\{1,\ldots,n\}} \left\{ AB_{index(c_i)} \vee \left(\left(\bigwedge_{c'\in ref(c_i)} c' \right) \rightarrow c_i \right) \right\}$$

where the function *index(c)* returns a unique index for each cell and *ref*(c) returns all cells that are referenced by cell c. A test case $T = (I, O)$ is transformed into

$$\mathrm{OBS}_D(T) = \bigcup_{(x,v)\in I} \{x == true\} \cup \bigcup_{(x,v)\in O'} \{x == true\} \cup \bigcup_{(x,v)\in O''} \{x == false\}$$

where $O' \subseteq O$ contains the output cells with correct values and $O'' \subseteq O$ contains the output cells whose computed values differ from their expected values. More details on the transformation into a dependency-based model and a variation of the dependency-based model can be found in [44].

Example 13.3 (Dependency-based model). The running example from Fig. 13.1 translates to $\mathrm{SD}_D = \{\mathrm{AB}_{\mathrm{B5}} \vee ((\mathrm{B2} \wedge \mathrm{B3} \wedge \mathrm{B4}) \rightarrow B5), \mathrm{AB}_{\mathrm{B6}} \vee (\mathrm{B5} \rightarrow \mathrm{B6}), \ldots\}$. The test case from Example 13.1 translates to $\mathrm{OBS}_D = \{\mathrm{B2}{==}\mathrm{true}, \ldots, \mathrm{B6}{==}\mathrm{true}, \mathrm{C6}{==}\mathrm{true}, \mathrm{D6}{==}\mathrm{false}, \mathrm{E2}{==}\mathrm{false}\}$.

In the **qualitative deviation model**, we reason about the information whether values are correct or if they should be greater or smaller. For example, if cell A3 = A1 + A2 and we know that the expected value in A1 is greater than the actually computed value, then the value in A3 should also be greater. Figure 13.2(a) continues this line of reasoning for the operators $+, *, -, /$. These tables can be easily transformed into the tables illustrated in Fig. 13.2(b) which can be directly used by constraint solvers. The transformation of a formula expression $e_3 = e_1 \odot e_2$ in cell c_i with $\odot \in \{+, -, *, /\}$ works as follows:

- If both e_1 and e_2 are numbers, then constr = $\mathit{tableConstr}(\mathit{AB}_{\mathit{index}(c_i)}, =, =, e, \mathit{table}(\odot))$,
- if e_1 is a reference to a cell c' and e_2 is a number n, then constr = $\mathit{tableConstr}(\mathit{AB}_{\mathit{index}(c_i)}, c', =, e, \mathit{table}(\odot))$,
- if e_1 is a number n and e_2 is reference to a cell c', then constr = $\mathit{tableConstr}(\mathit{AB}_{\mathit{index}(c_i)}, =, c', e, \mathit{table}(\odot))$,
- if e_1 references c' and e_2 is a reference to c'', then constr = $\mathit{tableConstr}(\mathit{AB}_{\mathit{index}(c_i)}, c', c'', e, \mathit{table}(\odot))$

where the function *index(c)* returns a unique index for each cell, $\mathit{table}(\odot)$ returns t_1 for $\odot \in \{+, *\}$ and t_2 for $\odot \in \{-, /\}$, and $\mathit{tableContr}(ab_j, e_1, e_2, e_3, t_i)$ represents the constraint that looks up possible value combinations for ab_j, e_1, e_2, e_3 in table t_i (see Fig. 13.2(b) for a definition of t_1 and t_2). Whenever an expression consists of sub-expressions, auxiliary variables tmp_i are introduced with i being a consecutive number to make the variables unique. A test case $T = (I, O)$ is transformed into

$$\mathrm{OBS}_Q(T) = \bigcup_{(x,v) \in \{I \cup O'\}} \{\mathrm{eq}(x, =)\} \cup \bigcup_{(x,v) \in O''} \{\mathrm{eq}(x, <)\} \cup \bigcup_{(x,v) \in O'''} \{\mathrm{eq}(x, >)\}$$

where $\mathrm{eq}(x_1, x_2)$ is a constraint that ensures that x_1 is equal to x_2, $O' \subseteq O$ contains the output cells with correct values, $O'' \subseteq O$ contains the output

cells whose computed values are smaller than their expected values, and $O''' \subseteq O$ contains the output cells whose computed values are larger than their expected values. More information about the transformation of other operators can be found in [45].

e_1	$e_1 + e_2$ / $e_1 * e_2$ e_2	$<$	$=$	$>$
	$<$	$<$	$<$	?
	$=$	$<$	$=$	$>$
	$>$	?	$>$	$>$

x_1	$e_1 - e_2$ / e_1/e_2 e_2	$<$	$=$	$>$
	$<$	?	$<$	$<$
	$=$	$>$	$=$	$<$
	$>$	$>$	$>$	?

'?' stands for any value from $\{<, =, >\}$

(a) Qualitative deviation behavior

ab_{e_3}	e_1	e_2	e_3
false	$<$	$<$	$<$
false	$=$	$<$	$<$
false	$>$	$<$	$<$
false	$>$	$<$	$=$
false	$>$	$<$	$>$
false	$<$	$=$	$<$
false	$=$	$=$	$=$
false	$>$	$=$	$>$
false	$<$	$>$	$<$
false	$<$	$>$	$=$
false	$<$	$>$	$>$
false	$=$	$>$	$>$
false	$>$	$>$	$>$
true	.	.	.

ab_{e_3}	e_1	e_2	e_3
false	$<$	$<$	$<$
false	$<$	$<$	$=$
false	$<$	$<$	$>$
false	$=$	$<$	$>$
false	$>$	$<$	$>$
false	$<$	$=$	$<$
false	$=$	$=$	$=$
false	$>$	$=$	$>$
false	$<$	$>$	$<$
false	$=$	$>$	$<$
false	$>$	$>$	$<$
false	$>$	$>$	$=$
false	$>$	$>$	$>$
true	.	.	.

'.' stands for all values from $\{<, =, >\}$

(b) Tuple sets t_1 (left) and t_2 (right)

Fig. 13.2: Qualitative deviation behavior for $+, *.-, /$ and the derived tuples for the operators $+, *$ (t_1) and $-, /$ (t_2).

Example 13.4 (Qualitative deviation model). The running example from Fig. 13.1 has the qualitative deviation system description $\text{SD}_Q = \{\mathit{tableContr}(\text{AB}_{B5}, B3, B4, \text{tmp}_1, t_1),\ \mathit{tableContr}(\text{AB}_{B5}, \text{tmp}_1, B4, \text{tmp}_2, t_1), \mathit{tableContr}(\text{AB}_{B5}, \text{tmp}_2, =, \text{tmp}_3, t_2),\ \mathit{tableContr}(\text{AB}_{B5}, B2, B4, \text{tmp}_4, t_1), \mathit{tableContr}(\text{AB}_{B5}, \text{tmp}_3, \text{tmp}_4, B5, t_1),\ \ldots\}$. The test case from Example 13.1 translates to $\text{OBS}_Q = \{$(eq(B2,=), eq(B3,=), ..., eq(B6,=), eq(C6,=), eq(D6,<), eq(E2,<)$\}$.

13.4.3 *Solving mechanism*

Independent of the model type, the diagnoses can be computed either indirectly [41] or directly [43]. The indirect method first identifies the minimal conflict sets. Informally speaking, a conflict is a subset of the components

for which it can be concluded that not all of them can be working correctly given the observations.

Definition 13.2 (Conflict). A conflict for a diagnosis problem (COMP, SD, OBS) is a subset $\nabla \subseteq \text{COMP}$ where $\text{SD} \cup \text{OBS} \cup \{\neg\text{AB}(C) \mid C \in \nabla\}$ is not satisfiable. A conflict ∇ is minimal if and only if no proper subset of ∇ is a conflict.

Minimal conflict sets can be efficiently computed using the QuickXPlain [46] or MergeXPlain [47] algorithms. These minimal conflict sets can then be used to determine the diagnoses by means of Reiter's hitting set algorithm [39]. A hitting set contains at least one element of each conflict set.

Example 13.5 (Indirect diagnosis computation). For the value-based model from Example 13.2, we compute {D2, E2}, {C6, D2, D5, D6}, {B6, C2, C5, D2, D5, D6}, and {B5, C2, C5, D2, D5, D6} as minimal conflict sets. From these conflicts, we compute the minimal hitting sets {D2}, {D5, E2}, {D6, E2}, {C2, C6, E2}, {C5, C6, E2}, and {B5, B6, C6, E2} which are also the minimal diagnoses.

The direct method computes the minimal diagnoses sequentially after their cardinality as described in Algorithm 1: A constraint is added to the constraint system that limits the size of the computed diagnoses (Line 3). By definition, all supersets of diagnoses are also diagnoses. To prevent the reporting of these supersets, blocking clauses for each found diagnosis have to be added to the constraint system (Line 6) before diagnoses of higher cardinality are computed.

Example 13.6 (Direct diagnosis computation). For the value-based model from Example 13.2, the direct computation starts with searching for single-fault diagnoses. The solver returns one single fault diagnosis, in our case {D2}. A blocking clause for this diagnosis is added to the constraints and the solution size is set to two. The solver reports {D5, E2} and {D6, E2} as double fault diagnoses. After adding the blocking clauses for these diagnoses and increasing the solution size to three, the diagnoses {C2, C6, E2} and {C5, C6, E2} are found. After adding the blocking clauses and increasing the solution size, the solver returns {B5, B6, C6, E2} as a solution. This diagnosis is again added as a blocking clause to the constraint system. Increasing the diagnosis size further does not detect any other diagnoses.

Algorithm 1 ConDiag(M, COMP, n) (adapted from [48])

Input: System model M, components COMP
Output: All minimal diagnoses

1: diagnoses $= \{\}$
2: **for** $i = 1$ **to** $|\text{COMP}|$ **do**
3: $\quad \overline{M} = M \cup \left\{ \left(\sum_{j=1}^{|\text{COMP}|} \text{AB}[j] \right) == i \right\}$
4: $\quad S = \textbf{solve}(\overline{M})$
5: $\quad$ diagnoses $=$ diagnoses $\cup\, S$
6: $\quad M = M \cup \bigcup_{s \in S} \neg(s)$
7: **end for**
8: **return** diagnoses

Both the direct and the indirect methods rely on a solver for computing the diagnoses respectively conflict sets. Since the majority of the constraint solvers does not support real-valued numbers (e.g., Minion[5]) or does not support the reification of constraints containing real numbers (e.g. Choco[6]), [49] proposed to use the state-of-the-art SMT solver Z3.[7] The use of Z3 has not only made it possible to solve value-based models that contain real-valued numbers but also has significantly reduced the solving time compared to Choco and Minion.

Finding the faulty formulas is only the first step in the debugging process, where the ultimate goal is to fix (or: repair) the faulty formulas. Some approaches in the literature exist to compute repair operations from diagnoses. [50] use genetic programming techniques to determine which cells have to be changed in order to create repair candidates that satisfy the provided test case. Unfortunately, the number of such repair candidates can be large in this approach. Therefore, [51] proposed an alternative approach that uses the concepts of MBD to generate distinguishing test cases that filter out invalid repair candidates.

13.4.4 *Tool support*

The MS Excel add-in Exquisite [5] is based on the described MBD principles and supports users during the testing and debugging process. It

[5] `https://constraintmodelling.org/minion/`.
[6] `https://choco-solver.org/`.
[7] `https://github.com/Z3Prover`.

provides visual feedback about the different types of cells, i.e., input, intermediate, and output cells and users can create test cases by providing values for the input cells and expected values for output cells. Whenever a computed value differs from the expected value, EXQUISITE computes the diagnoses using the approach described before with a value-based model and indirect diagnosis computation. Then, users can either manually inspect the formulas provided by the diagnosis engine or create additional test cases which can be added to the CSP in order to narrow down the number of diagnoses [52].

Unfortunately, users may sometimes struggle to provide test cases for large spreadsheets because *(i)* it is difficult to provide input values which test all branches of a nested IF-formula when the IF-formula is far away from the input cells in the calculation chain, and *(ii)* it is cumbersome to provide input values for a large number of similar input cells. Therefore, [53] developed an extension of EXQUISITE that enables users to create test cases for smaller parts of a spreadsheet, so-called fragments. Similar to regular test cases, fragment test cases have input, intermediate and output cells, but the input cells need not to be real input cells, but could also be formula cells.

Besides the manual creation of fragment test cases, the tool extension can automatically split a spreadsheet into fragments using a genetic algorithm. The algorithm's fitness function considers the number of input and output cells, the area spanned by these cells, and the cells' formula complexities. The fragmentation process keeps two competing aspects in mind. On the one hand, the size of these automatically generated fragments should be small so that users can easily find input values that cover all branches of nested IF-formulas. On the other hand, the size of the fragment has to be large enough to keep logically connected cells together.

13.4.5 *Empirical evaluations and limitations*

Many of the model-based techniques discussed in this book chapter were mostly evaluated using small spreadsheets [42] or spreadsheets with artificially seeded faults such as the modified *EUSES* corpus [38]. [37] and [53], however, evaluated their approaches on spreadsheets with real faults, namely the *INFO1* and the *Enron Error* corpus. These evaluations show the effectiveness of model-based methods but also indicate that their usage in industrial applications may be hampered by the fact that they often return many diagnosis candidates. One approach to deal with such problem

is to use fragmentation techniques. An evaluation of the fragment-based spreadsheet debugging approach [53] using the *Enron Error* corpus showed that it significantly improved the debugging process by narrowing down the number of diagnoses and also the number of manually inspected formulas.

The fragmentation and test case creation process of this approach have been successfully evaluated in a user study. The main outcome of the study was that the fragmentation approach enabled users to efficiently and effectively test their spreadsheets. To the best of our knowledge, none of the other model-based spreadsheet debugging approaches so far was evaluated by means of a user study. However, uses studies are important because they can reveal unrealistic assumptions about the debugging process in general and the capabilities of users for handling such an approach, as the seminal study of [54] in the software engineering domain has shown. In addition, automated methods for test case creation and evaluation, similar to [55, 56] for general software artifacts, should be investigated to speed up and simplify testing procedures.

13.5 Benchmarks

The availability of representative collections of faulty spreadsheets is an important prerequisite for the development and evaluation of debugging approaches. Over the last decades, researchers created a number of such corpora, which were widely used in the approaches presented in this chapter. In this section, we provide an overview of the most important ones. Their details are summarized in Table 13.2.[8]

EUSES [20] and *Fuse* [57] are collections of spreadsheets obtained via Internet search. They contain spreadsheets from different domains that vary in size and structure. However, these spreadsheets have no annotated faults and are mostly used by researchers to derive corpora that contain fault information. For example, a modified version of the EUSES corpus presented by [38] contains 576 spreadsheets, where each spreadsheet comprises at least one known fault randomly inserted using mutation operators.

The *Enron* spreadsheet corpus contains more than 15,000 spreadsheets that were attached to emails sent by employees of Enron [58]. The *Enron Errors* corpus lists 36 faults in 31 of these spreadsheets [36]. Since the Enron spreadsheets are extracted from email attachments, [59] created the *VEnron* corpus that comprises more than 12,000 spreadsheets in 1,609 clusters,

[8] See `https://spreadsheets.ist.tugraz.at/index.php/corpora-for-benchmarking/` for further information and download links.

Table 13.2: Comparison of spreadsheet corpora.

Corpus	**Spreadsheets**	**Origin**	**Faults**
EUSES	4498	Internet search	Unknown
Mod. EUSES	576	EUSES corpus	Seeded
Fuse	249,376	Internet search	Unknown
Enron	15,770	Email attachments	Unknown
Enron Errors	31	Enron corpus	Real
VEnron	12,254	Enron corpus	Unknown
INFO1	119	Student assignment	Real
Hawaii Kooker	75	Student assignment	Real
Payroll/Gradebook	2	User study	Seeded
Mod. P/G	349	Payroll/Gradebook	Seeded
Integer	231	Real-life + artificial	Seeded

where each cluster comprises different versions of the same spreadsheet.

The *Hawaii Kooker* corpus [31] and the *INFO1* [37] collection comprise spreadsheets with real faults that were created by business and civil engineering students, resp., using Excel to solve practical problems. Faults that were found in these spreadsheets are indicated in the meta-information.

The *Payroll/Gradebook* corpus consists of versions of two small spreadsheets with five faulty cells that were used in a user study [60]. The original spreadsheets were written in Form/3, but Excel versions of these spreadsheets are also available [61].

Finally, the *Integer* corpus is a collection of 21 real-life and 12 artificially created spreadsheets. There are faulty versions of each spreadsheet with randomly seeded faults. The special feature of this corpus is that it contains only integer values [49].

13.6 Summary and Outlook

The discussions in the previous sections show that both main paradigms of modern AI research — machine learning and model-based reasoning — as well as heuristic-based approaches can be successfully applied for supporting the spreadsheet debugging process. An important aspect in that context is that many of the proposed solutions were evaluated on the basis of real-world spreadsheets. Despite this success and recent progress, there are still a number of important directions for future work in this area. Next, we will discuss three main areas for future research.

Development of Spreadsheet Fault Corpora. Today's research in algorithmic spreadsheet debugging is mostly evaluated on a small number of collections containing faults. These faults are either artificially injected, e.g., by using mutation operators [62], or they are real faults made by users like in the collection provided in [36]. While the existing corpora, listed in Table 13.2, are diverse in terms of their creators and the types of faults that can be found in them, larger collections would be desirable in particular from the perspective of approaches that are based on fault patterns and machine learning.

Furthermore, like in the *Mining Software Repositories* field in general software engineering, one problem of such corpora is that many faults that are made by the developers might have been corrected before the software is versioned. Therefore, many faults that were made on the way might not be present anymore in the repositories or the released spreadsheets. As a result, further research — maybe in the form of laboratory experiments — is required to understand which types of temporary faults users make and if specific debugging aids are required to capture these faults.

Algorithmic Improvements. From an algorithmic perspective, further improvements are possible, both for machine learning approaches and for model-based techniques. In particular, for the latter set of approaches, a deeper investigation of *qualitative reasoning* methods seems promising. Instead of reasoning with exact numerical values, qualitative reasoning approaches, as discussed above, can help us, for example, to work with intervals or bounds. In some cases, when observing a problem in a spreadsheet, users might not be immediately able to inform the debugger about the exact expected outcome for a certain cell. However, they might feel comfortable with providing a lower bound for the expected value in a cell.

Generally, many of today's AI-based debugging techniques assume a "one-shot" interaction paradigm, where the debugger — given a filled spreadsheet and potentially additionally test cases — determines and highlights a set of potentially faulty cells for the user to inspect. Such a set of fault candidates can, however, be large, and might involve dozens of suspicious cells. In the context of traditional model-based diagnosis techniques, a common approach to deal with such problems is to make additional "measurements". In a debugging scenario, this translates into interactively asking the user questions, e.g., about the correctness of a certain cell. While algorithmic approaches for sequential fault localization already exist [63], they are often designed to minimize the number of measurements. In a

debugging context, one has however also to consider the complexity of answering the question for the end-user, which has not been investigated in the literature before.

Human-Debugger Interface. Finally, and probably most importantly, many questions are open regarding the design of appropriate user interfaces of debugging tools for spreadsheets. A particular problem in the context of spreadsheets is that users can be diverse in terms of their technical background and their capabilities. For more advanced and technically-skilled users, comparably complex user interfaces (embedded in MS Excel) like proposed in [5, 53] can be still manageable from the perspective of their cognitive complexity. However, some less-experienced spreadsheet users might be overwhelmed by such a system, even when it is more or less seamlessly embedded in their usual spreadsheet environment. For such users, novel lightweight and self-explanatory interaction mechanisms are needed, e.g., different forms of visualizing the "smelliness" of a certain formula or subarea of the spreadsheet.

In general, more user studies are needed to understand the real value of AI-based spreadsheet debugging tools. In the field of general software engineering, [54] questioned if purely data-based analyses are sufficient to answer the question: do debuggers really help software engineers. One outcome of their study was that some of the core assumptions of algorithmic debugging actually do not seem to hold in practice and that some approaches might be less useful as one might guess from data-based experiments. While previous user studies indicate that also more complex debugger interfaces can be lead to increased fault localization performance [64] for experienced users, it is unclear if these more complex interaction mechanisms would be useful also for users that are not well-versed in the use of spreadsheets.

Acknowledgments

The work was funded by the Austrian Science Fund (FWF) project *Interactive Spreadsheet Debugging* (iDEOS) under contract number P 32445-N38.

References

[1] R. R. Panko, Thinking is bad: Implications of human error research for spreadsheet research and practice, *CoRR* **abs/0801.3114** (2008), `arXiv:0801.3114`.

[2] D. Jannach, T. Schmitz, B. Hofer and F. Wotawa, Avoiding, finding and fixing spreadsheet errors - A survey of automated approaches for spreadsheet QA, *Journal of Systems and Software* **94**, pp. 129–150 (2014).
[3] P. Koch, K. Schekotihin, D. Jannach, B. Hofer and F. Wotawa, Metric-based fault prediction for spreadsheets, *IEEE Transactions on Software Engineering* **forthcoming** (2019).
[4] R. Singh, B. Livshits and B. Zorn, Melford: Using neural networks to find spreadsheet errors, *Microsoft Technical Report MSR-TR-2017-5* (2017).
[5] D. Jannach and T. Schmitz, Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach, *Automated Software Engineering* **23**, 1, pp. 105–144 (2016).
[6] W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, A survey on software fault localization, *IEEE Transactions on Software Engineering* **42**, 8, pp. 707–740 (2016).
[7] D. Radjenovic, M. Hericko, R. Torkar and A. Zivkovic, Software fault prediction metrics: A systematic literature review, *Information & Software Technology* **55**, 8, pp. 1397–1418 (2013).
[8] C. Catal, Software fault prediction: A literature review and current trends, *Expert Systems with Applications* **38**, 4, pp. 4626–4636 (2011).
[9] M. Erwig and M. M. Burnett, Adding apples and oranges, in S. Krishnamurthi and C. R. Ramakrishnan (eds.), *4th International Symposium on Practical Aspects of Declarative Languages*, pp. 173–191 (2002).
[10] Y. Ahmad, T. Antoniu, S. Goldwater and S. Krishnamurthi, A type system for statically detecting spreadsheet errors, in *18th IEEE International Conference on Automated Software Engineering*, pp. 174–183 (2003).
[11] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth and M. Felleisen, Validating the unit correctness of spreadsheet programs, in *26th International Conference on Software Engineering*, pp. 439–448 (2004).
[12] R. Abraham and M. Erwig, UCheck: A spreadsheet type checker for end users, *Journal of Visual Languages & Computing* **18**, 1, pp. 71–95 (2007).
[13] C. Chambers and M. Erwig, Automatic detection of dimension errors in spreadsheets, *Journal of Visual Language & Computing* **20**, 4, pp. 269–283 (2009).
[14] W. Dou, S. Cheung and J. Wei, Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation, in *36th International Conference on Software Engineering*, pp. 848–858 (2014).
[15] W. Dou, C. Xu, S. C. Cheung and J. Wei, CACheck: Detecting and repairing cell arrays in spreadsheets, *IEEE Transactions on Software Engineering* **43**, 3, pp. 226–251 (2017).
[16] L. Xu, S. Wang, W. Dou, B. Yang, C. Gao, J. Wei and T. Huang, Detecting faulty empty cells in spreadsheets, in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 423–433 (2018).
[17] S.-C. Cheung, W. Chen, Y. Liu and C. Xu, CUSTODES: automatic spreadsheet cell clustering and smell detection using strong and weak features, in *38th International Conference on Software Engineering*, pp. 464–475 (2016).
[18] D. Li, H. Wang, C. Xu, F. Shi, X. Ma and J. Lu, WARDER: Refining Cell

Clustering for Effective Spreadsheet Defect Detection via Validity Properties, in *19th International Conference on Software Quality, Reliability and Security*, pp. 139–150 (2019a).

[19] D. Li, H. Wang, C. Xu, R. Zhang, S.-C. Cheung and X. Ma, SGUARD: A Feature-Based Clustering Tool for Effective Spreadsheet Defect Detection, in *34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1142–1145 (2019b).

[20] M. Fisher II and G. Rothermel, The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms, *ACM SIGSOFT Software Engineering Notes* **30**, 4, pp. 1–5 (2005).

[21] D. W. Barowy, E. D. Berger and B. G. Zorn, ExceLint: automatically finding spreadsheet formula errors, *Proceedings of the ACM on Programming Languages* **2**, pp. 148:1–148:26 (2018).

[22] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez and J. Saraiva, Smelling faults in spreadsheets, in *30th IEEE International Conference on Software Maintenance and Evolution*, pp. 111–120 (2014).

[23] F. Hermans, M. Pinzger and A. van Deursen, Detecting and visualizing inter-worksheet smells in spreadsheets, in *34th International Conference on Software Engineering*, pp. 441–451 (2012a).

[24] F. Hermans, M. Pinzger and A. van Deursen, Detecting code smells in spreadsheet formulas, in *International Conference on Software Maintenance*, pp. 409–418 (2012b).

[25] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes and J. Saraiva, Smellsheet detective: A tool for detecting bad smells in spreadsheets, in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 243–244 (2012a).

[26] J. Cunha, J. P. Fernandes, H. Ribeiro and J. Saraiva, Towards a catalog of spreadsheet smells, in *12th International Conference on Computational Science and Its Applications*, pp. 202–216 (2012b).

[27] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez and J. Saraiva, FaultySheet Detective: When Smells Meet Fault Localization, in *30th IEEE International Conference on Software Maintenance and Evolution*, pp. 625–628 (2014).

[28] A. A. d. S. Meyer, A. A. F. Garcia, A. P. d. Souza and C. A. L. d. Souza Jr., Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L), *Genetics and Molecular Biology* **27**, pp. 83–91 (2004).

[29] R. Abreu, P. Zoeteweij and A. J. C. van Gemund, An evaluation of similarity coefficients for software fault localization, in *12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46 (2006).

[30] Lucia, D. Lo, L. Jiang, F. Thung and A. Budi, Extended comprehensive study of association measures for fault localization, *Journal of Software: Evolution and Process* **26**, 2, pp. 172–219 (2014).

[31] S. Aurigemma and R. R. Panko, The detection of human spreadsheet errors by humans versus inspection (auditing) software, in *EuSpRIG 2010 Conference*, pp. 1–14 (2010).

[32] I. J. Goodfellow, Y. Bengio and A. C. Courville, *Deep Learning*. MIT Press (2016).
[33] C. Cortes and V. Vapnik, Support-vector networks, *Machine Learning* **20**, 3, pp. 273–297 (1995).
[34] R. E. Schapire, A brief introduction to boosting, in *16th International Joint Conference on Artificial Intelligence*, pp. 1401–1406 (1999).
[35] L. Breiman, Random forests, *Machine Learning* **45**, 1, pp. 5–32 (2001).
[36] T. Schmitz and D. Jannach, Finding errors in the enron spreadsheet corpus, in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 157–161 (2016).
[37] E. Getzner, B. Hofer and F. Wotawa, Improving spectrum-based fault localization for spreadsheet debugging, in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 102–113 (2017).
[38] B. Hofer, A. Riboira, F. Wotawa, R. Abreu and E. Getzner, On the empirical evaluation of fault localization techniques for spreadsheets, in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, pp. 68–82 (2013).
[39] R. Reiter, A theory of diagnosis from first principles, *Artificial Intelligence* **32**, 1, pp. 57–95 (1987).
[40] R. Dechter, *Constraint processing*. Elsevier Morgan Kaufmann (2003).
[41] D. Jannach and U. Engler, Toward model-based debugging of spreadsheet programs, in *9th Joint Conference on Knowledge-Based Software Engineering*, pp. 252–262 (2010).
[42] R. Abreu, A. Riboira and F. Wotawa, Debugging Spreadsheets: A CSP-based Approach, in *International Workshop of Program Debugging (IWPD) at ISSREW 2012*, pp. 159–164 (2012).
[43] R. Abreu, B. Hofer, A. Perez and F. Wotawa, Using constraints to diagnose faulty spreadsheets, *Software Quality Journal* **23**, 2, pp. 297–322 (2015).
[44] B. Hofer, A. Höfler and F. Wotawa, Combining models for improved fault localization in spreadsheets, *IEEE Transactions on Reliability* **66**, 1, pp. 38–53 (2017a).
[45] B. Hofer, I. Nica and F. Wotawa, Qualitative deviation models for spreadsheet debugging, in *International Workshop on Program Debugging (IWPD) at ISSRE*, pp. 191–198 (2017b).
[46] U. Junker, QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems, in *Proceedings of the 19th National Conference on Artificial Intelligence / IAAI*, pp. 167–172 (2004).
[47] K. Shchekotykhin, D. Jannach and T. Schmitz, MergeXplain: Fast computation of multiple conflicts for diagnosis, in *24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pp. 3221–3228 (2015).
[48] I. Nica and F. Wotawa, ConDiag - computing minimal diagnoses using a constraint solver, in *23rd International Workshop on Principles of Diagnosis*, pp. 185–191 (2012).
[49] S. Außerlechner, S. Fruhmann, W. Wieser, B. Hofer, R. Spork, C. Mühlbacher and F. Wotawa, The Right Choice Matters! SMT Solving Substantially Improves Model-Based Debugging of Spreadsheets, in *13th International Conference on Quality Software*, pp. 139–148 (2013).

[50] B. Hofer and F. Wotawa, Mutation-based spreadsheet debugging, in *International Workshop on Program Debugging at ISSRE*, pp. 132–137 (2013).

[51] R. Abreu, S. Außerlechner, B. Hofer and F. Wotawa, Testing for Distinguishing Repair Candidates in Spreadsheets - the Mussco Approach, in *27th International Conference on Testing Software and Systems*, pp. 124–140 (2015).

[52] T. Schmitz and D. Jannach, An AI-based interactive tool for spreadsheet debugging, in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 333–334 (2017).

[53] D. Jannach, T. Schmitz, B. Hofer, K. Schekotihin, P. W. Koch and F. Wotawa, Fragment-based spreadsheet debugging, *Automated Software Engineering* **26**, 1, pp. 203–239 (2019).

[54] C. Parnin and A. Orso, Are automated debugging techniques actually helping programmers? in *2011 International Symposium on Software Testing and Analysis*, pp. 199–209 (2011).

[55] R. Almaghairbe and M. Roper, Automatically classifying test results by semi-supervised learning, in *ISSRE*. IEEE Computer Society, pp. 116–126 (2016).

[56] M. Roper, Using machine learning to classify test outcomes, in *AITest*. IEEE, pp. 99–100 (2019).

[57] T. Barik, K. Lubick, J. Smith, J. Slankas and E. Murphy-Hill, Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets, in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 486–489 (2015).

[58] F. Hermans and E. Murphy-Hill, Enron's spreadsheets and related emails: A dataset and analysis, in *37th International Conference on Software Engineering, ICSE '15*, Vol. 2, pp. 7–16 (2015).

[59] W. Dou, L. Xu, S. Cheung, C. Gao, J. Wei and T. Huang, Venron: A versioned spreadsheet corpus and related evolution analysis, in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 162–171 (2016).

[60] J. R. Ruthruff, M. Burnett and G. Rothermel, Interactive fault localization techniques in a spreadsheet environment, *IEEE Transactions on Software Engineering* **32**, 4, pp. 213–239 (2006).

[61] E. Getzner, *Improvements for Spectrum-based Fault Localization*, Master's thesis, Graz University of Technology (2015).

[62] R. Abraham and M. Erwig, Mutation operators for spreadsheets, *IEEE Transactions on Software Engineering* **35**, 1, pp. 94–108 (2009).

[63] K. Schekotihin, T. Schmitz and D. Jannach, Efficient sequential model-based fault-localization with partial diagnoses, in *25th International Joint Conference on Artificial Intelligence*, pp. 1251–1258 (2016).

[64] P. Rodler, D. Jannach, K. Schekotihin and P. Fleiss, Are query-based ontology debuggers really helping knowledge engineers? *Knowledge-based Systems* **179**, pp. 92–107 (2019).

This page intentionally left blank

https://doi.org/10.1142/9789811239922_0014

Chapter 14

Artificial Intelligence Methods for Software Debugging

Wolfgang Mayer[a] and Franz Wotawa[b]
[a] *University of South Australia, Adelaide, Australia*
[b] *Graz University of Technology, Graz, Austria*

14.1 Introduction

Software systems can be difficult to develop and maintain due to their complexity and sheer size. As software code is written, faults may be introduced, some of which may not be found until later when users report undesirable software behavior. Developers of popular software systems receive hundreds of bug reports each day and expend considerable effort on understanding the reported issue, diagnosing its root cause, and developing fixes. As debugging and resolving problems in software consumes a significant fraction of the resources devoted to software development, automated technologies that can help improve the efficiency and effectiveness of software debugging have been desired for some time.

Creating intelligent support tools that can support software engineers with debugging tasks has been an active topic of research since the 1980's. Early approaches were mainly based on hand-crafted algorithms and heuristics designed to minimize some measure of effort for isolating faults in software code. More recently, artificial intelligence techniques have been introduced in automated debugging assistants.

Artificial intelligence techniques for debugging span a variety of tasks. They can help design tests to aid discovering faults, complement code analysis techniques by identifying patterns that are indicative of (in)correct use of libraries and interfaces, exonerate parts of code that is likely correct,

highlight code fragments that are likely faulty, and focus the developer's attention by ranking the remaining candidate explanations. In some cases, artificial intelligence techniques can even synthesize corrections that replace faulty code. The underlying artificial intelligence technologies include mining patterns from past test runs and program versions, counterfactual reasoning about the effects of code fragments to infer explanations from observations, use of natural language processing techniques to extract information from bug trackers and code repositories, and machine learning approaches capable of generating and ranking candidate explanations.

In this chapter we survey artificial intelligence techniques for software debugging. Our focus will be on debugging, excluding test generation and program repair techniques. We begin with a general outline of the role of artificial intelligence techniques in the software debugging context, present two common approaches for localizing the origin of a fault in the program's source code and summarize machine learning approaches aiming to support the human software developer. We conclude by summarizing the capabilities of the current methods and avenues for further research.

14.2 AI in the Debugging Life Cycle

Software developers seeking to find and resolve faults in complex programs generally engage in a sequence of activities that aim to detect faults, investigate the root causes of detected faults, document faults and their causes, and repair the program to resolve the faults. Figure 14.1 illustrates a view of this process. The activities highlighted in bold are the focus of this chapter.

Fault detection is usually achieved through testing, where the program is exercised on one or more test cases that aim to trigger a fault in the program, or through analysis of the source code of the program, which can be done manually or with the help of automated program analysis tools. Once a fault has been detected, its root cause must be determined, and relevant elements of the program's source code associated with the root cause identified. Once that is accomplished the fault has been localized in the source code. The outcomes of the diagnostic activities can be captured

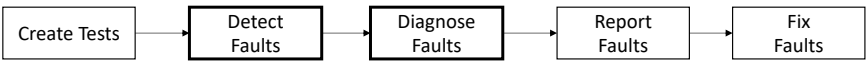


Fig. 14.1: Activities in the Debugging Life Cycle.

in a repository, for example, as reports in an issue tracking system. Once the root cause of the fault has been localized, the program can be modified to resolve the fault, and the test and detection regime can be amended to catch similar faults that may arise as the program evolves. The associated changes to the source code are generally captured in a version control system.

Tools that can partly automate this process and provide intelligent support to software developers have been sought for decades, as the complexity of software and number of potential faults therein has rendered purely manual investigations infeasible. The opportunities for using artificial intelligence techniques to augment the debugging process can be summarized as follows:

Create tests. Automated test generation mechanisms have been developed that can synthesize test cases that can detect and isolate specific faults in large programs, and methods for optimizing the collective efficiency of test cases in large test suites have been developed. Automated methods to isolate failure inducing changes [1], test suite minimization [2] and prioritization [3], and test generation [4] are well-known methods. [5] presented a survey of related techniques in this field.

Detect faults. Learning methods can infer patterns from execution traces that help discriminate normal program states from abnormal program states to detect errors within a program execution and help isolate where in a program execution an incorrect program state may have been entered. Moreover, pattern mining methods can be applied to source code to infer code patterns that are indicative of faults in programs. Selected techniques for fault detection are presented in Sec. 14.3.

Diagnose faults. Artificial intelligence techniques for localizing, ranking, and isolating faults based on reasoning about execution traces and possible faults can separate the parts of the program which may contain a fault from those that are not consistent with the observed program behavior. These techniques can reduce the human effort in fault localization. Model-based reasoning and spectrum-based candidate ranking approaches are two prominent techniques for debugging. Section 14.4 describes selected techniques in detail. Intelligent assistants for manual debugging can learn from historic debugging activity to guide human debugging tasks and help transfer debugging-related knowledge among software developers. Section 14.4.3

summarizes selected artificial intelligence and machine learning methods that can support debugging activities.

Report faults. Software repositories and related issue tracking systems provide a rich source of data that can be exploited to complement code-based analysis (see [6, 7]). Data mining of bug report texts and associated program code can yield predictors of fault proneness in software. Such predictions are useful to guide testing effort and to rank candidate faults identified in the diagnostic phase of debugging.

Fix faults. Automated program repair techniques have emerged that strive to automatically correct faulty programs. Methods include automated patching of program states at runtime to yield more resilient executions [8], synthesis of candidate patches based on mining historic source code patches [9], and automated synthesis of corrections using evolutionary, sub-symbolic, and other AI techniques. Monperrus [10] and Gazzola and colleagues [11] survey the state of the art in automated program repair techniques.

14.3 Fault Detection Techniques

Automated testing is one of the most widely used techniques for detecting faults in programs. The effectiveness of testing depends on the quality of the underlying test suite, and generating concise test suites that effectively detect a variety of failures can be prohibitively expensive for complex programs. The approaches to test suite generation and optimization are too numerous to present them here. Instead, we refer the reader to related literature on this topic. Nie and Leung [12] presented an overview of combinatorial test generation techniques, and Campos and colleagues [13] discussed evolutionary approaches to test suite optimization. Moreover, information about remaining candidate root causes in a debugging setting can be used to generate additional tests that may help discriminate between the candidate causes [14].

Although testing can determine that programs exhibit faults, identifying which part of a program is the root cause for observed test failures can be difficult since test runs can span many parts of a program. Assertions embedded within a program can detect earlier that a program execution has entered an incorrect state and help narrow the scope of the search for the bug. However, selecting appropriate invariants that are effective for

revealing faults can be difficult, and manually adding many invariants can be tedious. Instead, machine learning methods can be utilized to learn properties of programs that may indicate problems based on either the source code ("static analysis" techniques) or the execution traces of programs ("dynamic analysis" techniques).

Static analysis techniques for identifying "code smells" that may be indicative of incorrect use of language features are commonplace in many development environments. For example, *FindBugs* [15] highlights potential problems in Java source code based on anti-pattern rules. Machine learning methods can automatically learn models of correct programming interface usage from code examples. Deviations from the learned protocol model can then indicate potential problems in a program and can serve as additional signals for fault localization. Xie and Pei [16] used frequent sequence mining techniques to learn correct usage patterns based on a set of example code fragments demonstrating the use of an object, whereas Wasylkowski and colleagues [17] presented an approach that learns state machines that capture the correct sequences of function invocations. Machine learning techniques for code clone detection can also be employed to identify potentially incorrect or vulnerable code based on known instances of incorrect program fragments [18].

Invariant learning methods can infer invariant expressions that hold at a point in the program for all program executions resulting in passed test cases while violating some of the failing test cases [19, 20]. Since these methods exploit runtime information about programs, they belong to the dynamic analysis category. Invariants can include properties, such as value ranges for individual variables, relationships among multiple variables, application-specific predicates, and relations among object structures. Violations of invariants can indicate potential problems earlier in the execution than the ultimate test outcome. Abreu and colleagues [21] demonstrated that learned invariants can be included as additional signals to improve spectrum-based fault localization (SBFL) methods even if only simple range and bit mask invariants are employed.

Hybrid approaches that encompass elements of static, dynamic, heuristic, and machine learning methods are best positioned to yield robust detection of potential problems in complex programs. Ghaffarian and Shahriari [22] present a review of such approaches for vulnerability detection.

14.4 Fault Localization Techniques

Algorithmic and automated approaches for locating faults in programs have been of interest for several decades. Ehud Shapiro [23] was one of the first providing an algorithmic solution for supporting programmers fixing bugs in their Prolog programs. Mark Weiser [24, 25] analyzed the way programmers manually search for bug fixes and provided a theoretical basis when introducing the concept of static slices. Although, the general debugging problem seems to be always the same comprising a program, and a set of test cases where at least one test case is a failing one, debugging support may vary depending on the programming languages and the experience of the involved programmers. For experienced programmers we may consider different techniques than for programming novices were for the latter programming tasks are usually well defined. Murray [26] utilized this difference in his approach for guiding programming novices during their assigned tasks. In this section, we focus on the foundations behind programming tools for debugging supporting and automated debugging for more experienced programmers.

In particular, we discuss the two currently most popular approaches to fault localization in programs, i.e., model-based debugging and spectrum-based fault localization. According to Wong and colleagues [27] more than 50% of all published research articles on debugging make use of one of these two debugging techniques. We introduce the techniques making use of a small program `foo`. In Fig. 14.2 we depict the source code of method `foo`. This program takes two variables, `x` and `s`, as inputs and computes the area `a` and circumference `c` of a square (if `s` is 0) or a circle (if `s` different from 0). Depending on the value of `s` the other input variable `x` represents either the length of a side of the square, or the radius of the circle. When knowing the equations for the area and circumference of squares or circles it is obvious that there is a bug in Line 6 of `foo`. However, in the case of fault localization, we usually rely on test cases. So let us assume a set of test cases for `foo` that is given in Table 14.1.

Obviously, `foo` fails on test cases TC_3 and TC_5 where a correct result is computed for `a` but an incorrect result is derived for `c`. Hence, there is at least one failing test case and we are interested in computing the root cause of this observation. In principle, there are two more general ways of providing a solution to this debugging problem. First, we may consider information derived from the source code. For example, we know that variable `a` assigned in lines 5 and 8 always depends on the input variable

```
1.  public void foo (float x, int s) {
2.     float a;
3.     float c;
4.     if (s == 0) {
5.       a = x * x;
6.       c = 3 * x; // Bug!  Should be c = 4 * x;
7.     } else {
8.       a = 3.14 * x * x;
9.       c = 3.14 * 2.0 * x;
10.    }
11. }
```

Fig. 14.2: A small program fragment implementing a method `foo`.

Table 14.1: A set of test cases for method `foo`.

	`x`	`s`	`a`	`c`	passing
TC_1	0.0	0	0.0	0.0	√
TC_2	0.0	1	0.0	0.0	√
TC_3	1.0	0	1.0	4.0	×
TC_4	1.0	1	3.14	6.28	√
TC_5	2.0	0	4.0	8.0	×
TC_6	2.0	1	12.56	12.56	√

`x`. The same holds for variable `c`. Hence, when `c` is considered wrong, we only need to make use of these dependencies to collect all statements, one of which may be wrong. Second, we may make use of all available test cases together with their concrete executions. In particular, we know that when executing test case TC_3 lines 1, 2, 3, 4, 5, and 6 are executed (ignoring lines only comprising closing parentheses). When executing TC_4 lines 1, 2, 3, 8, and 9 are executed. Hence, considering the executions and the information whether a test case is passing or failing, we are able to assign a certain likelihood of being correct to each statement. The model-based debugging approach discussed in following section adopts the first principle, whereas the spectrum-based approach discussed in the subsequent section adopts the second principle.

It is also worth noting that more recently the use of machine learning for fault localization has been investigated [28–30]. These techniques mainly focus on learning use patterns or syntax patterns in the source code that

often lead to trouble. Making use of past programs such patterns can be learned and further used for locating suspicious code fragments.

14.4.1 *Model-based fault localization*

Model-based reasoning [31–33] is a method originating from artificial intelligence that allows to compute diagnoses directly from a model of a system comprising structural properties, such as components and their interconnections, and the components' behavior. A diagnosis is derived from the model of the system and a given set of observations that may contradict the expected behavior of the program. Diagnoses are computed using health assumptions about the components. If a component is assumed to be in the health state *correct*, the behavior of the component is equivalent to the one described in the model for the component. Otherwise, the behavior of the component is not specified. Using these health assignments, we are able to determine those components that need to behave *abnormally* in order to retract any inconsistency.

When adopting model-based reasoning for fault localization in programs, the main challenge is to find an appropriate model of the program we want to debug. Console and colleagues [34] provided the first step towards the objective of utilizing model-based reasoning for debugging. In their seminal paper, they considered logic programs, where the logical model used for reasoning is the program itself, and showed that the resulting debugger improves Shapiro's algorithmic debugging [23]. Bond [35] improved the work of Console and colleagues in his Ph.D. thesis. Liver [36] introduced a different approach requiring users to provide a model of a program, which is less appealing when considering that programs can be large and that we wish to automate the computation of diagnoses. Therefore, all other papers dealing with *model-based fault localization* (also called *model-based debugging*) rely on Console *et al.*'s paper where the model can be constructed automatically.

In this section, we describe in detail how general programs can be converted into models that can be directly used for debugging. We follow the definitions of Wotawa and colleagues [37] where the authors propose to map imperative programs to constraints. In Fig. 14.3 we summarize the debugging approach. The first step is to convert a program Π into a representation $\mathcal{L}(\Pi)$ using a compiler. The representation $\mathcal{L}(\Pi)$ is such that a theorem prover or constraint solver can compute a set of diagnoses Δ_S from the model and a failing test case TC. Each diagnosis $\Delta \in \Delta_S$ corresponds

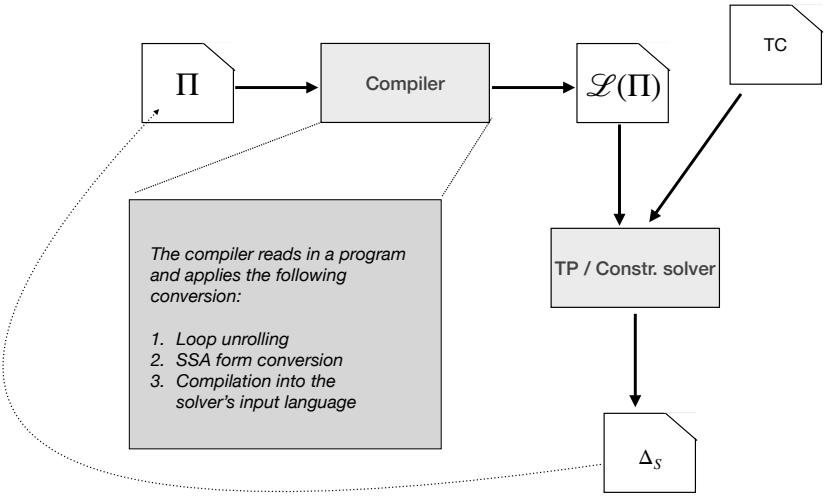


Fig. 14.3: The model-based fault localization flow chart.

to statements or expressions in the program Π that must all be faulty to explain the behavior of the given failing test case.

We now discuss the compilation step in more detail. The underlying idea behind the compilation is to create a model of the program, that is, a formal representation of the program that can be used for computing diagnoses. Here, we utilize a constraint representation where a constraint is an equation that defines a relationship between variables. To represent a program as a set of constraints we make use of the following observation: An assignment statement such as `a = x * x;` states that the variable `a` will have the value resulting from the multiplication of variable `x` with itself. This relationship among the two variables may be expressed as a constraint $a = x \cdot x$ where the constraint variable a corresponds to the program variable `a`, and constraint variable x to program variable `x`, respectively.

Therefore, we are able to map assignment statements such as `a = x * x;` more or less directly to a constraint, while paying close attention to capturing the sequential nature of statements. Consider two consecutive statements `a = x * x; a = 1;`. Regardless of the value of program variable `x`, the value of `a` after executing both statements in the given sequence is 1. Mapping both statements to equations would lead to a set of equations $\{a = x \cdot x, a = 1\}$ also constraining the value of x to be 1 (or -1) as well, but this is not what we want to achieve. Our constraint representation

must consider the sequence of statements as well! To ensure this, we use the static single assignment form (SSA) of a program (see [38]). The SSA form assures that every variable is only defined once, i.e., any variable in a program is allowed to occur at most once at the left side of an assignment statement. For the conversion, variables are associated with an index i that is initially 0. Every time a variable is assigned, its index is incremented by 1. If the same variable is used in a subsequent expression, the new index is used. For example, the SSA form of the program fragment `a = x * x; a = 1;` is $\texttt{a}_1$ `=` $\texttt{x}_0$ `*` $\texttt{x}_0$`;` $\texttt{a}_2$ `= 1`. Mapping the SSA form of the program to a set of equations represents the program correctly.

The SSA also requires mapping conditional statements, such as if-then-else, and loops statements, such as while and repeat-until statements, to a specific representation. For if-then-else (ITE) statements the SSA conversion is as follows: For each assignment in the then- and else-block, we introduce a new assignment with distinct indices. At the end of the ITE statement, we introduce an additional assignment for every variable that was assigned in any block of the ITE statement, where on the right hand side a Φ function is used to select either the variable from the then-block or from the else-block depending on the value of the condition expression of the ITE statement. We expand ("unroll") loops into nested ITE statements, which yields a program that shares the same behavior as the original one provided that the number of considered loop iterations does not exceed the nesting depth of the ITE statements. For example, consider the statement `while` C { S }. This statement is converted into nested if-then-else statements `if` C { S `if` C { S `if` C { S ...} } }.

In summary, the conversion of a given program Π into a formal model comprises the following steps: (1) Eliminating all loops and recursive function calls by unrolling or similar techniques. (2) Converting the resulting loop-free program into its SSA form, and (3) mapping the SSA to a formal representation.

Example 14.1. The resulting SSA form of our method `foo` from Fig. 14.2 is as follows:

```
temp1 := (s0 == 0)
a1 := x0 * x0
c1 := 3.0 * x0
a2 := 3.14 * x0 * x0
c2 := 3.14 * 2.0 * x0
a3 := Φ(temp1, a1, a2)
c3 := Φ(temp1, c1, c2)
```

In this representation, the if-then-else statement (lines 4–10) is mapped to a set of constraints. `temp1 := (s1 == 0)` is used to capture the value of the conditional expression for later use in the Φ functions. The assignment statements for variables `a3` and `c3` make use of the Φ function to deliver new values for the original variables `a` and `c` respectively, after executing the if-then-else statement. □

It is worth noting that the SSA of a program only comprises assignment statements. The used indices guarantee to correctly represent the sequence of statements, and each SSA statement can be also traced back to its representation in the original program. What we have not discussed so far in detail is the conversion into a formal constraint representation and handling of the health states of certain program fragments. For representing a health state, we introduce a special variable `ab` for each statement of the original program and make use of this variable to switch on or off the behavior of the statement. For example, a statement 5. `a = x * x;` if correct, would establish a relationship between variables `a` and `x`, which can be represented as a constraint $a = x \cdot x$. However, in case the statement is regarded as faulty, this does not hold anymore. We can represent this using the following constraint $ab_5 \vee (a = x \cdot x)$ meaning that either statement 5 is abnormal (i.e., not correct) or it has its stated behavior. In general, for each statement i with corresponding constraint C_i, we create a constraint model $ab_i \vee C_i$. Note that for if-then-else statements we only use the ab variable when assigning a value to its corresponding condition.

Example 14.2. Taking care of the health states of statements, and the SSA of method `foo` given in Fig. 14.2, we are able to finally come up with a formal representation that can be directly used for fault localization. The SMT-LIB[1] model for `foo` used by constraint and SMT solvers such as Z3[2] looks as follows:

[1] See `http://smtlib.cs.uiowa.edu/language.shtml`.
[2] See `https://github.com/Z3Prover`.

```
(declare-fun x_0 () Real)
(declare-fun s_0 () Int)
(declare-fun temp_0_0 () Bool)
(declare-fun ab_0 () Bool)
(declare-fun a_1 () Real)
(declare-fun ab_1 () Bool)
(declare-fun c_1 () Real)
(declare-fun ab_2 () Bool)
(declare-fun a_2 () Real)
(declare-fun ab_3 () Bool)
(declare-fun c_2 () Real)
(declare-fun ab_4 () Bool)
(declare-fun a_3 () Real)
(declare-fun c_3 () Real)
(assert (or ab_0 (= temp_0_0 (= s_1 0))))
(assert (or ab_1 (= a_1 (* x_0 x_0))))
(assert (or ab_2 (= c_1 (* 3.0 x_0))))
(assert (or ab_3 (= a_2 (* 3.14 x_0 x_0))))
(assert (or ab_4 (= c_2 (* 3.14 2.0 x_0))))
(assert (= a_3 (ite temp_0_0 a_1 a_2)))
(assert (= c_3 (ite temp_0_0 c_1 c_2)))
(minimize (+
   (ite ab_0 1 0)
   (ite ab_1 1 0)
   (ite ab_2 1 0)
   (ite ab_3 1 0)
   (ite ab_4 1 0)))
(check-sat)
```

In this representation the health states of all assignments as well as the conditional statement are represented using the `ab_`i variables. □

In order to compute diagnoses, we have to specify a test case. For our model, we define values of input and the output variables. Input variables `v` use the constraint variable with index zero, whereas output variables use the largest index of each variable. The resulting diagnosis problem is that given a constraint model $\mathcal{L}(\Pi)$ of a program Π having statements $0, \ldots, n$, and a failing test case TC, search for a set of statements that explain the misbehavior. In model-based reasoning a diagnosis is usually defined as follows:

Definition 14.1. Given a constraint model $\mathcal{L}(\Pi)$ of a program Π, a set of statements $S = \{0, \ldots, n\}$, and a failing test case TC. A set $\Delta \subseteq S$ is a diagnosis if and only if $\mathcal{L}(\Pi) \cup TC \cup \{ab_i = \textbf{true} | i \in \Delta\} \cup \{ab_i = \textbf{false} | i \in S \backslash \Delta\}$ is consistent, i.e., satisfiable.

Using this definition fault localization becomes a search problem where we are interested in identifying a subset of statements that when assumed to behave incorrectly, render the model consistent with the given failing test case. For more information regarding diagnosis algorithms and their runtime performance we refer the interested reader to Nica and colleagues [39].

Example 14.3. Making use of the constraint representation given in Example 14.2 for method `foo`, the test case $TC = \{x_0 = 1, s_0 = 0, a_3 = 1, c_3 = 4\}$ (which has to be translated to fit the SMT-LIB format), and the SMT solver Z3, we obtain only one diagnosis of cardinality one, i.e., $\Delta_1 = \{6\}$. There is another diagnosis stating that the condition in the if-then-else statement is wrong together with the statements in lines 8 and 9. However, usually someone is only interested in obtaining diagnoses that are the smallest. □

Model-based reasoning has been used for diagnosis in various programming languages, including VHLD [40], Verilog [41], Java [42, 43], Logic programs [34, 35], knowledge bases [44], functional languages [45], and spreadsheets [46, 47]. Depending on the underlying programming language the models used for diagnosis vary. We may consider different program fragments as diagnosis components, where we assign health state variables. We may also consider the different programming language paradigms. For sequential programming languages, the modeling steps given in this book chapter are appropriate. For concurrent languages, we may not rely on the SSA form. For functional languages, we need to consider the way functions are called and how they return their results. For every programming language, the modeling has to be adapted to fit the underlying semantics. However, once fixed we can implement a compiler that converts every program into its corresponding model.

In addition to differences originating from the underlying programming languages and their paradigms, different models with varying characteristics can be created for a program. The model presented here captures the semantics in term of computing values of variables directly. However, we may also make use of abstractions like in [48] for handling loops without the need for unrolling. Moreover, we may only capture control or data dependencies for diagnosis (see [40]). Wotawa [49] showed that such simple dependency models provide the same diagnostic accuracy as static slicing [24]. In order to improve upon these results while still using only dependencies a model may consider using improved slicing methods such as dynamic slicing [50] to capture more precise dependencies.

14.4.2 *Spectrum-based fault localization*

SBFL is an automated debugging technique implementing the following underlying idea. Instead of making use of formal models derived from the source code, SBFL solely relies on differences in program execution traces induced by a set of test cases. An execution trace for a given test case is a set of statements that are executed when running the program using the input values specified in the test case. The set of statements in a trace are sometimes called *coverage*. Execution traces for passing and failing test cases affect the likelihood of statements being faulty. For example, consider the case where a particular statement is only executed in failing runs but never in passing ones. We would immediately conclude that such a statement is very likely to be a root cause. Consider another statement

which is never executed in failing runs. Such a statement can hardly be the root cause of a test failure. Unfortunately, there are many statements that are executed in passing and in failing runs; hence, we need a way to ascribe a likelihood of being faulty to these statements. In SBFL, a suspiciousness index quantifies the likelihood of a statement being a root cause.

Jones and Harrold [51] were the first to develop the foundations underlying SBFL. Using their Tarantula system, which was mainly working at the module level rather than the level of individual statements, programmers were able to visualize the suspiciousness of certain program fragments by mapping suspiciousness index information to a color space. Later Janssen and colleagues [52] described another SBFL system based on a different computation of suspiciousness at the level of statements. In all these cases it is necessary to have a set of test cases where at least one test case is a failing one, and the ability to extract execution traces or coverage. It is worth noting that coverage information, i.e., information whether a statement is executed in a certain run, is sufficient for computing the suspiciousness index.

In Fig. 14.4 we summarize the overall SBFL process starting with a program Π where we apply annotations allowing us to obtain execution traces leading to a modified program Π'. The annotation basically are program fragments that are executed when a corresponding statement is executed and storing this information. Program Π' is executed on every given test case and the trace information is extracted. This information is

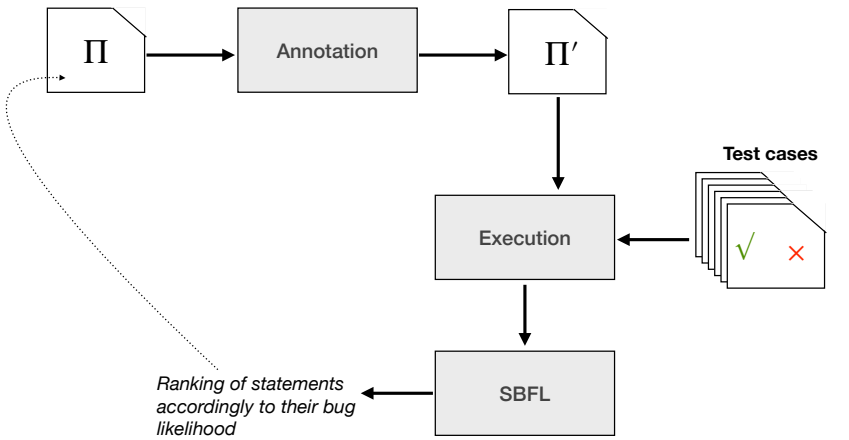


Fig. 14.4: The SBFL flow chart.

used to come up with a suspiciousness ranking of statements of the original program Π that is returned to the user for inspection. In the following, we outline the computation of the suspiciousness index and illustrate the foundations of SBFL.

Given a set of test cases TS, we first log all statements of a program that are executed in a certain test case run. We obtain a matrix called *observation matrix* where each row represents a statement i and each column a test case j. An entry o_{ij} in the observation matrix O is 1 if statement i is executed when running test case j, and 0 otherwise. In addition to the observation matrix we consider an *error vector* E. An entry e_j of E is 1 if test case j is a failing test case, i.e., the program output differs from the expected one, and 0 if test case j is a passing test case.

Example 14.4. In Fig. 14.5 we depict whether a certain test case stated in Table 14.1 executes a statement in method `foo` from Fig. 14.2. For example, test case 1 (TC_1) is a passing test case where statements 2, 3, 4, 5, and 6 are executed. Note that we do not include non-executable code fragments (lines 1, 7, 10, 11).

		Test cases					
		1	2	3	4	5	6
1.	`public void foo (float x, int s) {`						
2.	`  float a;`	×	×	×	×	×	×
3.	`  float c;`	×	×	×	×	×	×
4.	`  if (s == 0) {`	×	×	×	×	×	×
5.	`  a = x * x;`	×		×		×	
6.	`  c = 3.0 * x; // Bug!`	×		×		×	
7.	`  } else {`						
8.	`  a = 3.14 * x * x;`		×		×		×
9.	`  c = 3.14 * 2.0 * x;`		×		×		×
10.	`  }`						
11.	`}`						
	PASS	×	×		×		×

Fig. 14.5: The execution traces for the test cases $TC_1, \ldots, TC_6$ of method `foo`.

The observation matrix for `foo` is:

$$O_{\texttt{foo}} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

where the 1st row represents statement 2 and so forth. Given the information of which test case is a passing and which is a failing test case, we are able to obtain the error vector for `foo` as well:

$$E_{\texttt{foo}} = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Next, we consider how to obtain a ranking of statements considering their likelihood to be faulty. □

For computing a suspiciousness index for each statement from which we obtain a ranking, we make use of the observation matrix O and the error vector e. We first, compute values a_{kl} for each statement i where the computed value indicates whether the given statement is executed or not in a passing/failing test case. a_{kl} is defined as follows:

$$a_{kl}(i) = |\{j | j \in TS \wedge o_{ij} = k \wedge e_j = l\}|$$

The meaning of the indicators a_{kl} is as follows: a_{11} is the number of failing test cases where the statement was executed, a_{10} is the number of passing test cases where the statement was executed, a_{01} is the number of failing test cases where the statement was not executed, and finally a_{00} is the number of passing test cases where the statement was not executed. The values $a_{kl}(i)$ can now be used to compute the suspiciousness index $s(i)$ for every statement i, that is, $s(i) = f(a_{11}(i), a_{10}(i), a_{01}(i), a_{00}(i))$. It is worth noting that there are many such functions $f(.)$ defined in literature including the following:

$$s_{\text{O}}(i) = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \cdot (a_{11} + a_{10})}} \qquad \text{(Ochiai [53])}$$

$$s(i) = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} \qquad \text{(Jaccard [54])}$$

$$s_{\text{t}}(i) = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}} \qquad \text{(Tarantula [51])}$$

$$s_{\mathrm{S}}(i) = \frac{a_{11}}{a_{11} + a_{10}} \qquad \text{(State bug isolation [55])}$$

For all these definitions the case of a zero denominator is handled assuming a suspiciousness index (also called similarity coefficient) to be zero. The rationale behind these coefficients is to try score statements based on whether or not they were executed and if they were executed in more failing than passing test runs.

Example 14.5. When using the Ochiai coefficient, we are able to provide a ranking for the statements in program `foo`. We first compute the values for $a_{kl}(i)$ for each statement i. Afterwards, we compute the suspicious index s_{O} for the statements, and rank the statements accordingly. Statements with the same index share the same rank. Statements with the highest index s_{O} are assigned to rank 1. The ones with the next lower index to rank 2, and so forth. In Fig. 14.6 we summarize the values and rank for each statement. □

	a_{11}	a_{10}	a_{01}	a_{00}	s_{O}	R
1. `public void foo (float x, int s) {`						
2. `  float a;`	2	4	0	0	0.57	2
3. `  float c;`	2	4	0	0	0.57	2
4. `  if (s == 0) {`	2	4	0	0	0.57	2
5. `  a = x * x;`	2	1	0	3	0.81	1
6. `  c = 3.0 * x; // Bug!`	2	1	0	3	0.81	1
7. `  } else {`						
8. `  a = 3.14 * x * x;`	0	1	2	3	0.0	3
9. `  c = 3.14 * 2.0 * x;`	0	1	2	3	0.0	3
10. `  }`						
11. `}`						

Fig. 14.6: The similarity coefficients and rankings for method `foo`.

Besides the original SBFL approach several extensions and combinations have been discussed in the scientific literature. This includes the combination of slicing and SBFL (see for example, [56] and [57]) but also the combination of SBFL and model-based diagnosis (see [58]). Abreu and colleagues [59] introduced Barinel, a hybrid algorithm for multiple-fault diagnosis where the suspiciousness indexes and probabilistic component models were combined within a Bayesian reasoning framework. There are also many papers including [53] and [60] that report on the comparison of

different definitions of suspiciousness indices and similarity coefficients. It is also worth noting that the quality of the obtained ranking depends on the structure of the program and the available test cases.

14.4.3 *Machine-learning based fault localization*

Machine learning methods are attractive for software analysis as they can produce models and predictions based on data and may require little human interaction. In the software debugging context, data can often be obtained automatically, by analyzing the program's source code, instrumentation and logging of the program execution and outcomes at runtime, mining version histories and associated bug report data. This data contains potentially useful signals that can be identified by machine learning and data mining methods and used for improving fault localization and debugging activities in general.

The field of machine learning for debugging is still relatively young compared to the established techniques described in previous sections. Many techniques have been proposed for assisting fault localization using a variety of learning algorithms and input data. Some approaches aim to directly predict the location of a bug from data, for example from test coverage data, whereas others aim to augment existing debugging techniques with additional inputs or process their outputs to better pinpoint the true root cause of a failure.

14.4.3.1 *Learning for fault localization*

Machine learning techniques for fault localization generally restate the debugging problem as a prediction task, where a model learns to predict either fault indicators or a suspiciousness ranking for program elements based on the input data. Input data can include coverage information, similarity assessments computed using other techniques, code complexity metrics, and other features related to the program under consideration. Wong and colleagues [27] discussed some of the early works on machine learning and data mining for fault localization.

Wong and colleagues [61, 62] proposed deep learning based models for learning the relationship between the test coverage data and the corresponding test outcome. The learned models were excited separately with coverage vectors for each individual statement in the program to obtain predicted test outcomes. The model's prediction can be interpreted as the likelihood that the statement is faulty. Zhang and colleagues [63] followed

the same idea and use a deep neural net to quantify the suspiciousness of each statement. In addition, backward slicing from a failed test output was applied to focus the ranking on potentially faulty statements. Dutta and colleagues [64] presented a hierarchical approach to fault localization where the initial root cause was identified at the function level using a deep neural net trained on function-level coverage data and function complexity metrics. A separate neural net was used to identify likely faulty statements within the function in a subsequent stage.

Zhang and colleagues [65] augmented SBFL with ideas from PageRank. In their approach, the spectra obtained from the test suite were recomputed using the PageRank algorithm to reflect the different contributions of each test. The resulting augmented spectra were aggregated at different levels of granularity and could yield better fault localization than the original spectra.

Dutta and colleagues [66] proposed to learn an ensemble classifier capable of identifying potentially faulty statements. The ensemble consisted of two SBFL methods and three neural net classifiers, and the scores were combined using a weighted sum approach.

Several probabilistic approaches for learning debugging models have been proposed. Zhang and Zhang [67] employed statistical relational learning to infer a joint model of fault indicators and coverage data, program structure, and prior fault information. The joint inference mechanism had the potential to outperform traditional statistical fault localization which considers likelihood of statements being fault independently. Thaller and colleagues [68] learned a probabilistic model for localizing incorrect program elements that may be introduced as the program is modified. Their probabilistic model was structured as a network that reflected the dependencies, computations, and data flows in a program. The model was trained using traces from executions of the correct version of the program to yield joint distributions of related variables. Deviations in the distributions between the trained model and the observed distributions of a faulty program were used to rank candidate explanations. Nath and Domingos [69] proposed to use the Tractable Probabilistic Models framework to learn models that reflected program dependency structures from a corpus of faulty programs. The approach had the potential to outperform simpler SBFL approaches due to its ability to account for dependencies between program elements.

Fault prediction methods that aim to quantify how likely it is that a program part may be at fault have been proposed to improve software diagnosis. The approach presented by Elmishali and colleagues [70] learned

to predict faultiness of each program element from features such as code complexity metrics and program change history. Software diagnosis was performed using a modification of the Barinel approach [59], where the confidence score obtained from this binary classification model subsequently replaced the uniform prior probability of a statement being faulty. Elmishali and colleagues [14] followed the same principle and presented an iterated debugging cycle where test generation, fault diagnosis, and fault prediction were iterated. In this model, the test generation and fault diagnosis tasks were guided by a fault predictor that is trained on the bug reports and change history of the program under consideration. The method prioritized the candidate root causes effectively, while it would need to be augmented with other techniques to effectively discard incorrect candidate explanations.

The aforementioned works predominantly focus on debugging within a single process. However, complementary techniques exist that seek to diagnose bugs manifested as problems in inter-process communication. Zaman and colleagues [71] presented an approach for isolating inter-process failure inducing system call sequences from operating system audit logs. Their approach rested on extracting segments of related system calls from the logs, applying principal component analysis, and detecting potential anomalies in the low-dimensional representation. Functions in application programs related to the abnormal sequences were isolated through an off-line profiling process, where sequences of system calls were associated with the application functions issuing them.

Zhou and colleagues [72] developed an approach for fault prediction in micro service orchestrations. Their approach used multiple classification models that were trained to identify faulty services and the type of fault that may have occurred in a service. The models were trained on a corpus of normal and abnormal test case executions, where the abnormal cases were created by fault injection. At runtime, the predictors continuously monitored the execution and predicted what fault (if any) may have occurred and in which service the fault may have occurred.

Gu and colleagues [73] presented a classification model for assessing if the root cause of a crashing test run resided in the stack trace at the time of failure. The classifier received as input the features extracted from selected stack frames and their associated functions and computes a binary outcome using a decision tree method. Although this approach could not directly pinpoint root causes in general, it has the potential to help focus the debugging activities on the relevant program parts.

14.4.3.2 *Learning for multiple-fault localization*

Debugging tools making use of the parsimony heuristic to eliminate unlikely root causes may be unsuitable for programs that bear multiple faults, as the assumption that passing test cases exonerate program statements may not be appropriate. Zakari and colleagues [74] reviewed debugging techniques related to multiple fault localization. Further selected works related to machine learning for multiple fault localization are summarized in this section.

Clustering techniques are commonly used to deal with multiple faults in a program [75]. Huang and colleagues [76] assessed the effectiveness of multiple clustering methods on fault localization and found that k-means clustering may outperform hierarchical clustering methods for fault localization. Briand and colleagues [77] trained a decision tree to partition test suites into clusters of tests that were likely to fail due to the same root cause. Their method used test inputs and outputs to partition the test suite. A separate fault localization method akin to Tarantula [78] was then applied to each partition to rank each program statement, and rankings were consolidated into an overall ranking for inspection. Gao and Wong [79] used a k-medoids clustering technique and a tailored distance metric to group test cases that may have been affected by the same faults, and derived rankings of candidate bug explanations from the failing test executions in each cluster and all successful test executions. The clusters could be examined in parallel to isolate different bugs at the same time.

Zheng and colleagues [80] adopted an evolutionary computation approach to finding multiple faults in programs. In their approach, a genetic algorithm operated on chromosomes that indicated which statements are considered to be faulty. A fitness function inspired by well-known suspiciousness metrics used in SBFL guided the evolutionary process to retain well performing chromosomes. According to the authors, this method outperformed SBFL if multiple faults were present.

14.4.3.3 *Learning to Rank*

Fault localization techniques that are based on the *learning to rank* principle [81] directly compute a ranked list of potentially faulty statements without first identifying a set of faulty program elements. In this approach, top-ranked elements are considered to be faulty whereas elements ranked much lower are considered to be less likely explanations of the observed failures. Machine learning techniques play an essential role in learning the ranking function underpinning this technique.

Wang and colleagues [82] built on the observation that no single metric outperforms all others in SBFL. An evolutionary search process synthesized composite metrics that minimize average rank of faults in the training corpus. Xuan and Monperrus [83] followed a similar idea where weights associated with each of a set of base similarity metrics were learned from pairs of faulty and correct statements. Sohn and Yoo [84] extended SBFL with code and change metrics, applied evolutionary programming methods to develop novel ranking formulas, and used support vector machines to learn pairwise ranking models. The authors showed that their extended models could outperform the underlying SBFL approaches.

Le and colleagues [85] proposed a multi-stage approach to computing rankings, which rested on clustering, invariant learning, and ranking model learning. First, functions were clustered by the passing test cases they participated in. For each cluster, invariants were learned three times using the Daikon [20] tool: for all runs, only passing runs, and only failing runs. The differences between the invariants learned for each group were supplied as features in addition to the suspiciousness scores calculated by different SBFL metrics sourced from the literature. A support vector machine (SVM) was then trained to rank faulty program elements higher than correct ones. Li and Zhang [86] combined mutation-based fault localization and SBFL techniques to determine the impact of mutations on test outcomes and applied a SVM model to learn rankings. Li and colleagues [87] relied on multiple neuronal nets in a deep learning framework to infer effective latent features and learn ranking models. This approach overcame the limitation of traditional learning to rank methods that may face difficulties scaling to the large number of input features arising from combining SBFL-based features, mutation-based features, code complexity features, and textual features.

14.4.3.4 *Learning to prioritize issues*

Developers of popular software systems can receive a large number of bug reports each day. As the number of reported bugs grow, identifying which reports should be investigated first can become challenging, and developers can easily be overwhelmed by the number of reports received. Artificial intelligence techniques can help analyze and prioritize these reports to reduce the burden on developers. Uddin and colleagues [88] discussed relevant bug prioritization literature and systems. Classification models including the Naive Bayes method, SVM, and clustering methods have been prevalent in

the analysis of natural language based bug report repositories. Tian and colleagues [89] improved classification significantly by introducing additional factors, including temporal, textual, author, related reports, severity, and software product information into the classification model.

Kim and colleagues [90] found that a small number of crashes could account for a large number of bug reports. To detect, at the time when a crash report is first received, if the crash is likely to attract a large number of reports in the future, a predictor using features extracted from code and the bug report was learned from bug reports received in previous releases of the same software. The results indicated that the top crashes could be identified correctly in more than 75% of cases.

Wang and Lo [91] addressed the problem of selecting the source code files that may contain a bug based on the contents of bug reports. Their technique integrated version history, similar reports, and the report's structure into an ensemble predicting a suspiciousness score for each file. The combined approach was shown to outperform the individual methods by a considerable margin.

Ashok and colleagues [92] considered the related problem of retrieving bug reports that may be related to an observed failure. Contextual data about the failure was used to retrieve relevant document which were then linked with version control and bug repositories. Software repository mining techniques and PageRank [93] were used to establish links among the document and repositories and recommend relevant source code, documents, and people to the user.

14.4.3.5 *Method Selection*

Choosing an appropriate debugging technique for a given debugging scenario among the many proposed fault localization approaches can be a daunting task. Le and Lo [94] addressed this problem by attempting to predict the effectiveness of several debugging tools in a given debugging setting. Characteristic features of program traces and properties of the suspiciousness scores computed by each debugging tool were input to a discriminative model that determined if a true root cause would likely be among the top 10 ranked fault candidates. According to the authors, the approach exhibited good recall but suffered from low precision.

Zou and colleagues [95] investigated the effectiveness and efficiency of a range of fault localization techniques on real faults. The results revealed that although individual techniques were effective on the benchmark cases,

combinations of techniques significantly outperformed any individual technique.

14.4.4 *Automated oracles*

Debugging techniques, whether machine learning based or using traditional algorithmic approaches, rely on labeled data for their inferences. For example, model-based and spectrum-based debugging both rely on a test oracle that determines if a program state or test run is correct. Acquiring this information can be expensive in terms of human input and computing resources if many tests are run. Gao and colleagues [96] proposed an approach to automating the test oracle using a predictive model, where similar tests were clustered and the outcome of unlabeled tests was predicted by transferring the labels from known test outcomes in the same cluster. The predicted test outcomes were subsequently used as input to fault localization methods. Using the predicted outcomes for bug localization showed superior results than using only the tests where verified known outcomes were available.

14.4.5 *Summary of Techniques*

The characteristics of the machine-learning based methods discussed in this chapter are summarized in Table 14.2. Column *Input Data* lists the data required to apply each method once it has been trained on the training data. Many approaches require either test coverage information in the form of an execution *spectrum*, or information extracted from the failing execution at runtime. Fault localization methods that employ other techniques as prerequisites also use *suspiciousness scores* and *code metrics* computed by these methods. Column *Output* lists the result computed by each method. Here, *fault indicator* vectors, which indicate which program elements may be the root cause of a test failure, and *suspiciousness scores*, which indicate how likely it is that a program element is a root cause, are common among the considered approaches. Column *Training Data* lists the training data required for each method. Many methods rely on a test suite and the associated test outcomes; some methods require information about known bugs that cause the test failures or information about change history and bug reports collected for the program under consideration or similar programs. Frequent inputs are *spectra*, that is, coverage information about each test execution, *test outcomes*, that is, information if each test passed or failed, and *known bugs*, that is, the location of the root cause(s) of the failing test

Table 14.2: Characteristics of debugging methods.

Publication	Input Data	Training Data	Output	Learning Method
Fault Localization				
[61]	spectrum	spectra, test outcomes	suspiciousness scores	Deep Learning
[62]	spectrum	spectra, test outcomes	suspiciousness scores	Deep Learning
[63]	spectrum	spectra, test outcomes	suspiciousness scores	Deep Learning + Slicing
[64]	spectrum	spectra, test outcomes	fault indicator	Deep Learning + hierarchical abstraction
[65]	spectrum	spectra, test outcomes	fault indicator	SBFL+PageRank
[66]	spectrum	spectra, test outcomes, suspiciousness scores	suspiciousness scores	Ensemble classifier
[67]	spectrum	spectra, test outcomes, known bugs	fault indicator	Stat.rel.learning
[68]	spectrum	spectra, test outcomes, program states	fault indicator	Prob. Network model
[69]	spectrum	faulty source code	fault indicator	Probabilistic model
[70]	spectrum	metrics (code & process), known bugs	fault indicator	SFL+Bayes+Classification
[14]	spectrum	metrics (code & process), known bugs	suspiciousness scores	Classification+SBFL+Bayes
[71]	audit log at point of crash	audit logs	failure inducing call sequences	Dimensionality reduction
[72]	execution trace	failing execution traces, injected fault	fault indicator	Classification
[73]	stack backtrace	stack traces, known bugs	fault indicator	Classification

Table 14.2: Characteristics of debugging methods (cont.)

Publication	Input Data	Training Data	Output	Learning Method
Fault Localization for Multiple Faults				
[75]	spectrum	spectra, test outcomes	suspiciousness scores	Clustering + SBFL
[76]	spectrum	spectra, test outcomes	suspiciousness scores	Clustering + SBFL
[77]	test suite	test inputs, test outcomes	suspiciousness scores	Decision Tree + SBFL
[79]	spectrum	spectra, test outcomes	suspiciousness scores	Clustering
[80]	spectrum	spectra, test outcomes	fault indicator	Evolutionary Algorithm
Learning to Rank				
[82]	spectrum	spectra, known bugs, suspiciousness scores	suspiciousness scores	Evolutionary Algorithm
[83]	spectrum, suspiciousness scores	spectra, known bugs, suspiciousness scores	suspiciousness scores	RankBoost
[84]	execution trace	execution traces, known bugs, code change metrics	suspiciousness scores	Evolutionary Algorithm + Classification
[85]	set of faulty programs	spectra, test outcomes, change history	suspiciousness scores	Clustering, Invariant learning, SBFL
[86]	spectra of mutants, test outcomes, suspiciousness scores	spectra of mutants, test outcomes, suspiciousness scores	suspiciousness scores	SBFL + Mutation + Classification
[87]	spectra of mutants, test outcomes, code metrics, suspiciousness scores	spectra of mutants, code metrics, test outcomes, suspiciousness scores	suspiciousness scores	Mutation + Deep Learning

Table 14.2: Characteristics of debugging methods (cont.)

Publication	Input Data	Training Data	Output	Learning Method
Bug Repository Analysis				
[89]	reported issue	issues, issue priority	issue prioritization	Classification
[90]	reported issue	issues, attention count	future attention	Classification
[91]	reported issue	issue, source code, change history	files containing bug	Classification
[92]	failing execution data	issues, bug reports, change history	relevant bug reports	PageRank
Method Selection				
[94]	spectrum, scores by debugging tools	spectra, scores by debugging tools	recommended debugging tool	Regression
[72]	scores by debugging tools	spectra, suspiciousness scores	recommended debugging tool	Classification
Automated Oracles				
[96]	spectrum	spectra, test outcomes	test outcome, suspiciousness scores	Clustering + SBFL

executions. Some methods utilize *suspiciousness scores* calculated by other methods, such as SBFL, and *code metrics* that aim to quantify various notions of complexity derived from the source code. Column *Learning Method* lists the inference algorithms used by each method. This aspect exhibits the most diversity among the considered approaches. A variety of learning algorithms are employed, including *Classification*, *Clustering*, *Deep Learning*, and combinations of these algorithms with other techniques, such as SBFL, mutation testing, hierarchical abstraction, and probabilistic inference methods.

14.5 Outlook

This chapter presented an introduction to the field of automated debugging and related artificial intelligence techniques for fault localization. The fault localization techniques based on source code analysis are among the mature techniques in the field, such that a variety of algorithms have demonstrated an effective reduction in effort required to find bugs in programs. There is opportunity for further research to utilize complementary information hidden in source code repositories, bug trackers, and other software engineering resources to further improve the capabilities of automated debugging systems. Artificial intelligence methods can help unlock the vast amount of human contributions and expertise hidden away in unstructured text discussions and uncover hidden patterns about design decisions, development processes, preferences, and even social interactions among developers and users to inform bug finding and resolution. Moreover, bringing together the achievements in automated debugging and automated program repair may foster advances beneficial to both research communities.

References

[1] A. Zeller and R. Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering* **28**, 2, pp. 183–200 (2002), doi:10.1109/32.988498.

[2] G. Rothermel, M. Harrold, J. Ostrin and C. Hong, An empirical study of the effects of minimization on the fault detection capabilities of test suites, in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 34–43 (1998), doi:10.1109/ICSM.1998.738487.

[3] S. Elbaum, A. Malishevsky and G. Rothermel, Test case prioritization: A family of empirical studies, *IEEE Transactions on Software Engineering* **28**, 2, pp. 159–182 (2002), doi:10.1109/32.988497.

[4] P. McMinn, Search-based software test data generation: A survey, *Software Testing, Verification and Reliability* **14**, 2, pp. 105–156 (2004), doi:10.1002/stvr.294.

[5] S. Yoo and M. Harman, Regression testing minimization, selection and prioritization: A survey, *Software Testing, Verification & Reliability* **22**, 2, pp. 67–120 (2012), doi:10.1002/stv.430.

[6] H. H. Kagdi, M. L. Collard and J. I. Maletic, A survey and taxonomy of approaches for mining software repositories in the context of software evolution, *J. Softw. Maintenance Res. Pract.* **19**, 2, pp. 77–131 (2007), doi:10.1002/smr.344.

[7] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. D. Backer and R. Haesen, Mining software repositories for comprehensible software fault prediction models, *J. Syst. Softw.* **81**, 5, pp. 823–839 (2008), doi:10.1016/j.jss.2007.07.034.

[8] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst and M. Rinard, Automatically patching errors in deployed software, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. ACM, Big Sky, Montana, USA, ISBN 978-1-60558-752-3, pp. 87–102 (2009), ISBN 978-1-60558-752-3, doi:10.1145/1629575.1629585.

[9] X. B. D. Le, D. Lo and C. L. Goues, History Driven Program Repair, in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1, pp. 213–224 (2016), doi:10.1109/SANER.2016.76.

[10] M. Monperrus, Automatic Software Repair: A Bibliography, *ACM Computing Surveys* **51**, 1, pp. 17:1–17:24 (2018), doi:10.1145/3105906.

[11] L. Gazzola, D. Micucci and L. Mariani, Automatic Software Repair: A Survey, *IEEE Transactions on Software Engineering* **45**, 1, pp. 34–67 (2019), doi:10.1109/TSE.2017.2755013.

[12] C. Nie and H. Leung, A survey of combinatorial testing, *ACM Computing Surveys* **43**, 2, pp. 11:1–11:29 (2011), doi:10.1145/1883612.1883618.

[13] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler and A. Arcuri, An empirical evaluation of evolutionary algorithms for unit test suite generation, *Information and Software Technology* **104**, pp. 207–235 (2018), doi:10.1016/j.infsof.2018.08.010.

[14] A. Elmishali, R. Stern and M. Kalech, DeBGUer: A Tool for Bug Prediction and Diagnosis, *Proceedings of the AAAI Conference on Artificial Intelligence* **33**, 01, pp. 9446–9451 (2019), doi:10.1609/aaai.v33i01.33019446.

[15] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler and J. Penix, Using Static Analysis to Find Bugs, *IEEE Software* **25**, 5, pp. 22–29 (2008), doi:10.1109/MS.2008.130.

[16] T. Xie and J. Pei, MAPO: Mining API usages from open source repositories, in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06. ACM, Shanghai, China, ISBN 978-1-59593-397-3, pp. 54–57 (2006), ISBN 978-1-59593-397-3, doi:10.1145/1137983.1137997.

[17] A. Wasylkowski, A. Zeller and C. Lindig, Detecting object usage anomalies, in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07. ACM, Dubrovnik, Croatia, ISBN 978-1-59593-811-4, pp. 35–44 (2007), ISBN 978-1-59593-811-4, doi: 10.1145/1287624.1287632.

[18] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam and B. Maqbool, A Systematic Review on Code Clone Detection, *IEEE Access* **7**, pp. 86121–86144 (2019), doi:10.1109/ACCESS.2019.2918202.

[19] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin, Dynamically discovering likely program invariants to support program evolution, in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99. ACM, Los Angeles, California, USA, ISBN 978-1-58113-074-4, pp. 213–224 (1999), ISBN 978-1-58113-074-4, doi:10.1145/302405.302467.

[20] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, The Daikon system for dynamic detection of likely invariants, *Science of computer programming* **69**, 1–3, pp. 35–45 (2007).

[21] R. Abreu, A. González, P. Zoeteweij and A. J. C. van Gemund, Automatic software fault localization using generic program invariants, in *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08. ACM, Fortaleza, Ceara, Brazil, ISBN 978-1-59593-753-7, pp. 712–717 (2008), ISBN 978-1-59593-753-7, doi:10.1145/1363686.1363855.

[22] S. M. Ghaffarian and H. R. Shahriari, Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey, *ACM Computing Surveys* **50**, 4, pp. 56:1–56:36 (2017), doi:10.1145/3092566.

[23] E. Shapiro, *Algorithmic Program Debugging*. MIT Press (1983).

[24] M. Weiser, Programmers use slices when debugging, *Communication of the ACM* **25**, 7, pp. 446–452 (1982).

[25] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering* **10**, 4, pp. 352–357 (1984).

[26] W. R. Murray, *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing (1988).

[27] W. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, A survey on software fault localization, *IEEE Transactions on Software Engineering* **42**, 8, pp. 707–740 (2016), doi:10.1109/TSE.2016.2521368.

[28] B. Livshits and T. Zimmermann, Dynamine: finding common error patterns by mining software revision histories, *ACM SIGSOFT Software Engineering Notes* **30**, 5, pp. 296–305 (2005).

[29] T. Menzies, J. Greenwald and A. Frank, Data mining static code attributes to learn defect predictors, *IEEE Trans. Software Eng.* **33**, 1, pp. 2–13 (2007), doi:10.1109/TSE.2007.256941.

[30] Z. Guan, X. Wang, W. Xin, J. Wang and L. Zhang, A survey on deep learning-based source code defect analysis, in *2020 5th International Conference on Computer and Communication Systems (ICCCS)*. IEEE, pp. 167–171 (2020).

[31] R. Davis, Diagnostic reasoning based on structure and behavior, *Artificial Intelligence* **24**, pp. 347–410 (1984).
[32] R. Reiter, A theory of diagnosis from first principles, *Artificial Intelligence* **32**, 1, pp. 57–95 (1987).
[33] J. de Kleer and B. C. Williams, Diagnosing multiple faults, *Artificial Intelligence* **32**, 1, pp. 97–130 (1987).
[34] L. Console, G. Friedrich and D. T. Dupré, Model-based diagnosis meets error diagnosis in logic programs, in *International Joint Conference on Artificial Intelligence (IJCAI)*. Chambery, pp. 1494–1499 (1993).
[35] G. W. Bond, *Logic Programs for Consistency-Based Diagnosis*, Ph.D. thesis, Carleton University, Faculty of Engineering, Ottawa, Canada (1994).
[36] B. Liver, Modeling software systems for diagnosis, in *International Workshop on Principles of Diagnosis (DX)*. New Paltz, NY, pp. 179–184 (1994).
[37] F. Wotawa, M. Nica and I.-D. Moraru, Automated debugging based on a constraint model of the program and a test case, *The journal of logic and algebraic programming* **81**, 4, pp. 390–407 (2012).
[38] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, *ACM Transactions on Programming Languages and Systems* **13**, 4, pp. 451–490 (1991).
[39] I.-D. Nica, I. Pill, T. Quaritsch and F. Wotawa, The route to success - a performance comparison of diagnosis algorithms, in *International Joint Conference on Artificial Intelligence*. AAAI Press, ISBN 978-1-57735-633-2, pp. 1039–1045 (2013), ISBN 978-1-57735-633-2.
[40] G. Friedrich, M. Stumptner and F. Wotawa, Model-based diagnosis of hardware designs, *Artificial Intelligence* **111**, 2, pp. 3–39 (1999).
[41] B. Peischl, N. Riaz and F. Wotawa, Automated debugging of VERILOG designs, *International Journal of Software Engineering and Knowledge Engineering* **22**, 05, pp. 695–723 (2012), doi:10.1142/S0218194012500209.
[42] C. Mateis, M. Stumptner and F. Wotawa, Modeling Java Programs for Diagnosis, in *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Berlin, Germany, pp. 171–175 (2000).
[43] W. Mayer, M. Stumptner, D. Wieland and F. Wotawa, Can ai help to improve debugging substantially? debugging experiences with value-based models, in *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. IOS Press, Lyon, France, pp. 417–421 (2002).
[44] A. Felfernig, G. Friedrich, D. Jannach and M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, *Artificial Intelligence* **152**, 2, pp. 213–234 (2004).
[45] M. Stumptner and F. Wotawa, Debugging Functional Programs, in *International Joint Conference on Artificial Intelligence (IJCAI)*. Stockholm, Sweden, pp. 1074–1079 (1999).
[46] D. Jannach and T. Schmitz, Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach, *Automated Software Engineering*, pp. 1–40 (2014), doi:10.1007/s10515-014-0141-7.
[47] D. Jannach, T. Schmitz, B. Hofer and F. Wotawa, Avoiding, finding and

fixing spreadsheet errors: A survey of automated approaches for spreadsheet QA, *Journal of Systems and Software* **94**, pp. 129–150 (2014).

[48] W. Mayer and M. Stumptner, Debugging program loops using approximate modeling, in *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Valencia, Spain, pp. 843–847 (2004).

[49] F. Wotawa, On the Relationship between Model-Based Debugging and Program Slicing, *Artificial Intelligence* **135**, 1–2, pp. 124–143 (2002).

[50] B. Korel and J. Laski, Dynamic Program Slicing, *Information Processing Letters* **29**, pp. 155–163 (1988).

[51] J. A. Jones and M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in *Proc. of the International Conference on Automated Software Engineering (ASE)*. ACM, pp. 273–282 (2005).

[52] T. Janssen, R. Abreu and A. J. C. van Gemund, Zoltar: A toolset for automatic fault localization, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, pp. 662–664 (2009).

[53] R. Abreu, P. Zoeteweij and A. J. c. Van Gemund, An evaluation of similarity coefficients for software fault localization, in *2006 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46 (2006).

[54] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, Pin-point: Problem determination in large, dynamic internet services, in *Proceedings of the IEEE International Conference on Dependable Systems and Networks* (2002).

[55] B. Liblit, M. Naik, A. X. Zheng, A. Aiken and M. I. Jordan, Scalable statistical bug isolation, in V. Sarkar and M. W. Hall (eds.), *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005*. ACM, pp. 15–26 (2005), doi:10.1145/1065010.1065014.

[56] B. Hofer and F. Wotawa, *Spectrum Enhanced Dynamic Slicing for better Fault Localization*, *ECAI*, Vol. 242. IOS Press, Netherlands, ISBN 978-1-61499-097-0, pp. 420–425 (2012), ISBN 978-1-61499-097-0.

[57] W. Wen, B. Li, X. Sun and J. Li, Program slicing spectrum-based software fault localization, in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Miami Beach, USA (2011).

[58] R. Abreu, W. Mayer, M. Stumptner and A. J. C. van Gemund, Refining spectrum-based fault localization rankings, in *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09. ACM, New York, NY, USA, ISBN 9781605581668, pp. 409–414 (2009a), ISBN 9781605581668, doi:10.1145/1529282.1529374.

[59] R. Abreu, P. Zoeteweij and A. J. van Gemund, Spectrum-Based Multiple Fault Localization, in *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 88–99 (2009b), doi:10.1109/ASE.2009.25.

[60] B. Hofer, A. Perez, R. Abreu and F. Wotawa, On the empirical evaluation of similarity coefficients for spreadsheets fault localization, *Automated Software Engineering* **22** (2015), doi:10.1007/s10515-014-0145-3.

[61] W. E. Wong and Y. Qi, Bp neural network-based effective fault localization, *International Journal of Software Engineering and Knowledge Engineering* **19**, 04, pp. 573–597 (2009), doi:10.1142/S021819400900426X.

[62] W. E. Wong, V. Debroy, R. Golden, X. Xu and B. Thuraisingham, Effective Software Fault Localization Using an RBF Neural Network, *IEEE Transactions on Reliability* **61**, 1, pp. 149–169 (2012), doi:10.1109/TR.2011.2172031.

[63] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng and X. Chang, Deep Learning-Based Fault Localization with Contextual Information, *IEICE Transactions on Information and Systems* **E100.D**, 12, pp. 3027–3031 (2017), doi:10.1587/transinf.2017EDL8143.

[64] A. Dutta, R. Manral, P. Mitra and R. Mall, Hierarchically Localizing Software Faults Using DNN, *IEEE Transactions on Reliability* , pp. 1–26 (2019), doi:10.1109/TR.2019.2956120.

[65] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang and S. Khurshid, An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank, *IEEE Transactions on Software Engineering* (2019), doi:10.1109/TSE.2019.2911283.

[66] A. Dutta, N. Pant, P. Mitra and R. Mall, Effective Fault Localization using an Ensemble Classifier, in *2019 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE)*, pp. 847–855 (2019), doi:10.1109/QR2MSE46217.2019.9021187.

[67] S. Zhang and C. Zhang, Software bug localization with markov logic, in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014. ACM, Hyderabad, India, ISBN 978-1-4503-2768-8, pp. 424–427 (2014), ISBN 978-1-4503-2768-8, doi:10.1145/2591062.2591099.

[68] H. Thaller, L. Linsbauer, R. Ramler and A. Egyed, Probabilistic Software Modeling: A Data-driven Paradigm for Software Analysis, *arXiv:1912.07936 [cs]* (2019).

[69] A. Nath and P. Domingos, Learning tractable probabilistic models for fault localization, in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16. AAAI Press, Phoenix, Arizona, pp. 1294–1301 (2016).

[70] A. Elmishali, R. Stern and M. Kalech, Data-augmented software diagnosis, in D. Schuurmans and M. P. Wellman (eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12–17, 2016, Phoenix, Arizona, USA*. AAAI Press, pp. 4003–4009 (2016).

[71] T. S. Zaman, X. Han and T. Yu, SCMiner: Localizing System-Level Concurrency Faults from Large System Call Traces, in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 515–526 (2019), doi:10.1109/ASE.2019.00055.

[72] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang and C. He, Latent error prediction and fault localization for microservice applications by learning from system trace logs, in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019. ACM, Tallinn, Estonia, ISBN 978-1-4503-5572-8, pp. 683–694 (2019), ISBN 978-1-4503-5572-8, doi:10.1145/3338906.3338961.

[73] Y. Gu, J. Xuan, H. Zhang, L. Zhang, Q. Fan, X. Xie and T. Qian, Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence, *Journal of Systems and Software* **148**, pp. 88–104 (2019), doi:10.1016/j.jss.2018.11.004.

[74] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed and R. A. Rasheed, Multiple fault localization of software programs: A systematic literature review, *Information and Software Technology* **124**, p. 106312 (2020), doi:10.1016/j.infsof.2020.106312.

[75] J. A. Jones, J. F. Bowring and M. J. Harrold, Debugging in Parallel, in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07. ACM, London, United Kingdom, ISBN 978-1-59593-734-6, pp. 16–26 (2007), ISBN 978-1-59593-734-6, doi:10.1145/1273463.1273468.

[76] Y. Huang, J. Wu, Y. Feng, Z. Chen and Z. Zhao, An empirical study on clustering for isolating bugs in fault localization, in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 138–143 (2013), doi:10.1109/ISSREW.2013.6688893.

[77] L. C. Briand, Y. Labiche and X. Liu, Using Machine Learning to Support Debugging with Tarantula, in *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. IEEE, Trollhattan, Sweden, ISBN 978-0-7695-3024-6, pp. 137–146 (2007), ISBN 978-0-7695-3024-6, doi:10.1109/ISSRE.2007.31.

[78] J. Jones, M. Harrold and J. Stasko, Visualization of test information to assist fault localization, in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 467–477 (2002), doi:10.1145/581396.581397.

[79] R. Gao and W. E. Wong, MSeer — An Advanced Technique for Locating Multiple Bugs in Parallel, *IEEE Transactions on Software Engineering* **45**, 3, pp. 301–318 (2019), doi:10.1109/TSE.2017.2776912.

[80] Y. Zheng, Z. Wang, X. Fan, X. Chen and Z. Yang, Localizing multiple software faults based on evolution algorithm, *Journal of Systems and Software* **139**, pp. 107–123 (2018), doi:10.1016/j.jss.2018.02.001.

[81] T.-Y. Liu, *Learning to Rank for Information Retrieval*. Springer (2011), ISBN 978-3-642-14266-6, doi:10.1007/978-3-642-14267-3.

[82] S. Wang, D. Lo, L. Jiang, Lucia and H. C. Lau, Search-based fault localization, in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 556–559 (2011), doi:10.1109/ASE.2011.6100124.

[83] J. Xuan and M. Monperrus, Learning to Combine Multiple Ranking Metrics for Fault Localization, in *IEEE International Conference on Software Maintenance and Evolution*, pp. 191–200 (2014), doi:10.1109/ICSME.2014.41.

[84] J. Sohn and S. Yoo, FLUCCS: Using code and change metrics to improve fault localization, in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017. ACM, Santa Barbara, CA, USA, ISBN 978-1-4503-5076-1, pp. 273–283 (2017), ISBN 978-1-4503-5076-1, doi:10.1145/3092703.3092717.

[85] T.-D. B. Le, D. Lo, C. Le Goues and L. Grunske, A learning-to-rank based

fault localization approach using likely invariants, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016. ACM, Saarbrücken, Germany, ISBN 978-1-4503-4390-9, pp. 177–188 (2016), ISBN 978-1-4503-4390-9, doi:10.1145/2931037.2931049.

[86] X. Li and L. Zhang, Transforming programs and tests in tandem for fault localization, *Proceedings of the ACM on Programming Languages* **1**, OOPSLA, pp. 92:1–92:30 (2017), doi:10.1145/3133916.

[87] X. Li, W. Li, Y. Zhang and L. Zhang, DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019. ACM, Beijing, China, ISBN 978-1-4503-6224-5, pp. 169–180 (2019), ISBN 978-1-4503-6224-5, doi:10.1145/3293882.3330574.

[88] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem and H. Shah, A survey on bug prioritization, *Artificial Intelligence Review* **47**, 2, pp. 145–180 (2017), doi:10.1007/s10462-016-9478-6.

[89] Y. Tian, D. Lo, X. Xia and C. Sun, Automated prediction of bug report priority using multi-factor analysis, *Empirical Software Engineering* **20**, 5, pp. 1354–1383 (2015), doi:10.1007/s10664-014-9331-y.

[90] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung and S. Park, Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts, *IEEE Transactions on Software Engineering* **37**, 3, pp. 430–447 (2011), doi:10.1109/TSE.2011.20.

[91] S. Wang and D. Lo, Version history, similar report, and structure: Putting them together for improved bug localization, in *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014. ACM, Hyderabad, India, ISBN 978-1-4503-2879-1, pp. 53–63 (2014), ISBN 978-1-4503-2879-1, doi:10.1145/2597008.2597148.

[92] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa and V. Vangala, DebugAdvisor: A recommender system for debugging, in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09. ACM, Amsterdam, The Netherlands, ISBN 978-1-60558-001-2, pp. 373–382 (2009), ISBN 978-1-60558-001-2, doi:10.1145/1595696.1595766.

[93] L. Page, S. Brin, R. Motwani and T. Winograd, The PageRank Citation Ranking: Bringing Order to the Web. (1999).

[94] T.-D. B. Le and D. Lo, Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools, in *2013 IEEE International Conference on Software Maintenance*. IEEE, pp. 310–319 (2013).

[95] D. Zou, J. Liang, Y. Xiong, M. D. Ernst and L. Zhang, An Empirical Study of Fault Localization Families and Their Combinations, *IEEE Transactions on Software Engineering*, pp. 1–1 (2019), doi:10.1109/TSE.2019.2892102.

[96] R. Gao, W. E. Wong, Z. Chen and Y. Wang, Effective software fault localization using predicted execution results, *Software Quality Journal* **25**, 1, pp. 131–169 (2017), doi:10.1007/s11219-015-9295-1.

This page intentionally left blank

https://doi.org/10.1142/9789811239922_bmatter

Index

2COMM, 58, 76

accountability, 80
Action, 312
active learning, 171
AdaBoost, 149
additive attention, 115
agent, 87, 311
Agent-oriented programming, 87
Agile, 307
agnostic framework, 319
AI techniques, 29, 32
algebraic structures, 202
analytic hierarchy process (AHP), 33
arrays of formula cells, 377
Artificial Intelligence (AI), 8, 308
Artificial Neural Networks (ANN), 308
Attention Mechanism, 114
attentional encoder-decoder model, 115
automate testing, 244
Automated program repair, 404
Automatic software generation, 169
autonomous agent, 87
Average Percentage of Faults Detected (APFD), 322

Background on Test Oracles
  sectional Test Oracles Structure Based on Test Cases
    subsection, 272
  sectional Test Oracles Structure Based on Test Data (Inputs)
    subsection, 273
Behavioral Programming, 3
BLEU, 116
Boundary test data generation, 361
business artifacts, 56
Business Process Model and Notation, 55
Business Processes, 55
business relevance, 33

Case Management Model and Notation, 56
Choco, 39
CI, 141, 142, 146, 308
clustering, 378
clusters, 320
cocircuits, 332, 344
code integration, 316
code reviews, 141–143, 149
coding guidelines, 142
coding rules, 145
combinatorial testing tools, 326
comparative analysis and envisionment, 333
Conceptual Integrity, 179
Conceptual-Disconnect, 177
Conflict, 389

conflict detection, 29
conflict sets, 41
conflicts, 38
consensus, 39
constraint programming, 384
constraint reasoning, 29
constraint solvers, 39
constraints, 32, 40
content-based recommendation, 42
Continuous Integration, 141, 308
convex polytope, 344, 353
coordination, 54
cost reduction, 319
covector, 343
coverage, 43
critical systems, 169
customer feedback, 317, 320

data analysis learning method, 310
data collection and analysis, 319
data type inference, 376
decision biases, 44
decision making, 44
decision support, 29
decision-making process, 307
Deep Learning, 170
Deep Neural Networks, 309, 381
defect clustering, 316
defect detection rate, 314
defect prediction, 309
dependencies, 30
dependency-based model, 386
design pattern, 180
Deterministic, 171
Diagnosis, 384
dimensionality reduction, 35
divide and conquer, 320

early requirements engineering, 30, 33
Eclipse, 44
eigenvalues/eigenvectors, 183
embedded system, 323
Encoder-Decoder Model, 113
envision graph, 355, 356
event, 315
evolutionary testing, 332
Execution Cycle, 319

fair, 39
fairness, 39
Fault detection, 402
Fault Localization, 406
fault prediction, 244
feature extraction, 148, 162
Fiedler vector, 183
finite domain variables, 40
fragmentation, 334, 337, 350, 356, 392
Functionals, 179

GCN (Graph Convolutional Networks), 170
genetic algorithm, 332
graph adjacency matrix, 188
Graph Auto-Encoders (GAE), 187
Graph Neural Networks (GNN), 187

heuristic procedures, 376
heuristic techniques, 308
hidden features, 36
higher-abstraction Software Design, 170
homodromies, 331, 337
homodromy graph, 356
Hybrid-DeepCom, 119
hyperplane arrangement, 343

Impacted(x), 311
inconsistencies, 41
intelligent Risk Based Analysis methodology (iRBA), 306
interest dimensions, 32, 33
Interoperability, 310
irba, 311
iRBA-3D-Graph, 312
iRBA-Dynamic-Scope, 311
iRBA-Link, 311
iRBA-Prioritization, 311
iRBA-Queue-Management iQ(), 311
iRBA-Reward, 311
iRBA-Reward-Functions, 311

JaCaMo, 76

Laplacian matrices, 170
Laplacian Spectral Verifier, 172
limit test data, 332, 334, 355, 362
Linear Software Models, 170
linkage criteria, 314
liquid-democracy, 34
loss, 29

Machine Learning (ML), 212–214, 219, 220, 231, 244, 308
Machine-learning based fault localization, 418
manual reviews, 142, 145
matrix factorization, 29, 36
METEOR, 117
minimal viable product (MVP), 31
minimum viable product, 36, 45
Model-based Diagnosis, 29, 383
Model-based reasoning, 403
Modern code reviews, 143
Modularity, 171
Modularity Matrix, 179
module Cohesion, 184
Modules-Sparsity, 185
Multiagent Systems, 51
multiple applications-under-test, 320
mutual clusters, 321
MVP, 36

Natural Language Processing (NLP), 309
next release problem, 38
Non-iRBA, 312
Non-standard analysis, 359

objective coordination, 63
online machine learning, 310
open source, 44
opportunity costs, 29
Optimal-non-iRBA, 322
optimization, 29, 32, 36, 212, 225, 227, 229, 230, 232, 312
Organizations, 57
oriented matroid, 331, 334, 337, 343
out-of-vocabulary (OOV) tokens, 118

pesticide paradox, 315
plan-creation process, 307
Pointer Generator, 124
post-release, 320
pre-release, 320
precedence logic, 58
preferences, 30, 38, 41
prioritization, 29–33, 36, 38, 43, 310, 312
probability, 314
product quality, 317
programming patterns, 54, 67
protocol, 52
PRSummarizer, 124
pull requests (PRs), 123

Q-learning, 310
qualitative deviation model, 387
qualitative execution, 360
qualitative model, 331, 364
qualitative reasoning, 331, 333, 334, 394
qualitative space algebra, 331
Quality Evaluation System, 321

real time, 310
real time analysis, 313
real-world environment, 322
recommendation, 29
Recurrent Neural Network (RNN), 113
regression, 308
Reinforcement Learning (RL), 309
release planning, 31, 37, 39, 40, 45
requirement, 29, 31, 38, 40, 42
requirement change request, 317
requirement review, 319
requirements engineering, 32
Resource saving, 325
resources, 317
review comments, 153
review process, 146
reward, 311
reward function, 315
Risk Assessment, 319
Risk Identification, 319

Risk-Based Testing (RBT), 306
RL Loss, 125
RoboCup, 88
robotics, 88
ROUGE, 117
RRGen, 129

Scenario-Based Programming, 5
Search-Based Software Engineering, 6
Sentiment analysis, 153
sequence-to-sequence (seq2seq) learning, 111
severity of the defect, 316
Shannon entropy, 379
similarity, 42
single applications-under-test, 320
social commitment, 52, 58
social networks, 32, 43
software artifact generation (SAG), 112
software concepts, 202
software correctness, 169
software debugging, 402
Software Design Generation, 170
Software Development Life Cycle (SDLC), 319
software development methodology, 8
Software Fusion, 170
software releases, 29
Software testing, 244
software testing principles, 315
spatial reasoning, 333
Spectral Verification, 171
Spectrum-based Fault Localization, 380, 405
Spreadsheet Fault Corpora, 394
spreadsheet smells, 379
stakeholder, 30, 32, 34, 35, 38, 39
stakeholder recommendation, 42
State, 312
Structors, 179
Structural-Based Traversal method (SBT), 120
subjective coordination, 57
super-covector, 337, 348
supervised learning, 381
Support Vector Machine (SVM), 309
suspiciousness measure, 380

Ternary Decision Tree, 331, 339
test case, 312
test case generation, 244
Test Case Prioritization (TCP), 308
test cases, 385
test cycle, 312
test data, 332
test execution, 313
Test Oracles Based on Machine Learning Techniques
  sectional Summary and Findings subsection, 284
  sectional Test Oracles Based on Supervised Learning Techniques subsection, 277
  sectional Test Oracles Based on Unsupervised Learning Techniques subsection, 283
  sectionalTest Oracles Based on Semi-Supervised Learning Techniques subsection, 280
Test Schedule, 319
test suite, 312
test suite reduction, 244
testing coverage, 313
testing quality, 317
Time to market (TTM), 322
transparency, 44
triage, 37

UML Class Diagram, 180
unit testing, 331–333, 364
unsupervised learning, 378
utility, 33
utility-based prioritization, 33

value-based constraint model, 386

weight, 315