Software Analysis and Refactoring Using Probabilistic Modelling and Performance Antipatterns

Ioannis Stefanakos

Doctor of Philosophy

University of York
Computer Science

December 2021

Abstract

In recent times, our reliance on software and software-controlled systems has drastically increased, as has the impact of failures in the operation of these systems. To guarantee the correct operation of software systems, it is necessary both to verify that they meet their functional requirements and to analyse their nonfunctional properties such as performance and reliability.

This thesis presents a novel two-pronged approach to the analysis and refactoring of software systems that must comply with strict nonfunctional requirements. The approach operates at both code-level and system architecture-level. At code-level, we use a new tool-supported method for the formal analysis of timing, resource use, and other quantitative aspects of the components of a software system. The new method synthesises a discrete-time Markov chain model of the analysed code, computes its transition probabilities using information from program logs, and employs probabilistic model checking to evaluate its performance properties of interest.

At system architecture-level, we use a new method that employs performance antipatterns, i.e., commonly occurring mistakes during software development with their solutions, and stochastic modelling to support any system refactoring that might be needed. Our method identifies the performance antipatterns present across the operational profile space of a software system, enabling engineers to detect operational profiles likely to be problematic for the analysed design, and supporting the selection of refactoring actions when performance requirements are violated.

The two methods are integrated into an end-to-end software performance engineering methodology that uniquely combines code-level probabilistic analysis with the use of performance antipatterns to guide refactoring.

We evaluate the proposed methods and the approach that integrates them using code obtained from several Java programs, including Android applications, and a foreign currency exchange service-based system comprising a combination of third-party and in-house components. Our results, subject to the case studies used in the evaluation, demonstrate the accuracy, efficiency and decision-making of the new approaches.

List of Contents

Abstract 3

List of Contents 5

List of Figures 9

List of Tables 11

Dedication 13

Acknowledgements 15

Declaration 17

1 Introduction 19

1.1 Motivation and Research Hypothesis 21
1.2 Contributions 23
1.3 Thesis Structure 24

2 Background 27

2.1 Model Checking 28
2.2 Probabilistic Model Checking 29
2.2.1 Markov Models 31
2.2.1.1 Discrete-Time Markov Chains 32
2.2.1.2 Continuous-Time Markov Chains 35
2.2.2 Probabilistic Temporal Logics 39
2.2.2.1 Probabilistic Computation Tree Logic 39
2.2.2.2 Continuous Stochastic Logic 42
2.2.3 Probabilistic Model Checking Tools and Applications 45
2.3 Performance Antipatterns 47

2.4 Summary 52

3 Probabilistic Analysis of Code Performance 53

3.1 Motivating Example 54
3.2 Approach 56
3.2.1 Probabilistic Model Extraction 58
3.2.2 Transition Probability Calculation 63
3.2.3 Probabilistic Model Checking 67
3.2.4 Further Application Scenarios 67
3.3 Implementation 68
3.4 Evaluation 68
3.4.1 Research Questions 68
3.4.2 Experimental Setup 69
3.4.3 Results and Discussion 70
3.4.4 Threats to Validity 74
3.5 Related Work 74
3.6 Summary 78

4 Software System Analysis and Refinement Using Performance Antipattern Profiles 79

4.1 Motivating Example 81
4.1.1 System Description 81
4.1.2 External Services 82
4.1.3 Internal Components 83
4.1.4 Operational Profile Parameters 83
4.2 Approach 84
4.2.1 Modelling 84
4.2.2 Model Instantiation 88
4.2.3 Model Analysis 89
4.2.4 Antipattern Profile Generation 91
4.2.4.1 Antipattern Detection Rules 91
4.2.4.2 Synthesis of Antipattern Profiles 94
4.2.5 Refactoring 95
4.3 Implementation 97
4.4 Evaluation 98
4.4.1 Research Questions 98
4.4.2 Experimental Scenarios 99
4.4.3 Experimental Results 100

4.4.4 Threats to Validity 106
4.5 Related Work 107
4.6 Summary 109

5 Software Performance Engineering with Code-level Probabilistic Analysis and Performance Antipatterns 111
5.1 Motivating example 112
5.2 Approach 114
5.2.1 Identification of System Components of Interest 115
5.2.2 Probabilistic Analysis of Code Performance 115
5.2.3 System Modelling 116
5.2.4 Antipattern-based System Analysis 116
5.2.5 Refactoring 117
5.3 Evaluation 118
5.3.1 Research Questions 118
5.3.2 Experimental Scenarios 118
5.3.3 Results and Discussion 123
5.3.4 Threats to Validity 128
5.4 Related Work 129
5.5 Summary 130

6 Conclusion and Further Research Directions 131
6.1 Conclusion 131
6.2 Further Research Directions 133
6.2.1 Probabilistic Analysis of Code Performance 133
6.2.1.1 Reliability Properties 133
6.2.1.2 Confidence Intervals for Performance Properties . . 134
6.2.1.3 Scalability 134
6.2.1.4 Practicality 134
6.2.2 Software System Analysis and Refinement Using Performance Antipattern Profiles 135
6.2.2.1 Trade-off Analysis 135
6.2.2.2 Portfolio of Generic Actions and Methods 135
6.2.3 Software Performance Engineering With Code-level Probabilistic Analysis and Performance Antipatterns 136
6.2.3.1 Additional Case Studies 136
6.2.3.2 Automation 136

Appendix A Chapter 3 - Supplementary Material 139

A.1 Java Subset Syntax Specification 139
A.2 Device Performance Case Study 144
A.3 Fast Sine Transformer Case Study 147
A.4 Knapsack Algorithm Case Study 150

Appendix B Chapter 4 - Supplementary Material **153**
B.1 Parametric DTMC Models 153
B.2 Parametric CTMC Model 157
B.3 Graphs for the CTMC Model's Properties 160

References **163**

List of Figures

2.1 State transition system of an alarm clock, adapted from [43] 29
2.2 Overview of probabilistic model checking, adapted from [123] 30
2.3 DTMC weather prediction model, adapted from [154] 34
2.4 A CTMC with four states modelling a queue of jobs, adapted from [134] 38
2.5 BLOB in industrial process control application, adapted from [144] . 49
2.6 Unbalanced processing caused by routing algorithm, adapted from [146] 51
2.7 Unbalanced processing in a P&F architecture, adapted from [146] . . 52

3.1 Java method `distance1` from the Apache Commons Math library . 55
3.2 PROPER program performance analysis 56
3.3 PROPER code-to-model transformation rules 58
3.4 DTMC model for the `distance1` Java method 59
3.5 PRISM model synthesised for the `distance1` Java method 63
3.6 Java method `power` on the left side of the figure, and its synthesised PRISM model representation on the right side. 66

4.1 Workflow of the foreign currency trading system (FX). The internal components of the system are depicted in blue colour and the external services in red. 81
4.2 Performance analysis and refactoring using antipattern profiles 84
4.3 Parameters of the parametric CTMC model (FX) 85
4.4 Request queue of the parametric CTMC model (FX) 86
4.5 Main workflow of the parametric CTMC model (FX) 87
4.6 Internal TA instance of the parametric CTMC model (FX) 88
4.7 Reward structures of the parametric CTMC model (FX) 88
4.8 Example of *antipattern profile* 94
4.9 BLOB antipattern instances while varying operational profiles. 101
4.10 CPS antipattern instances while varying operational profiles. 101
4.11 P&F antipattern instances while varying operational profiles. 102

4.12 BLOB antipattern instances across different refactorings - $scenario_A$. . 103
4.13 BLOB antipattern instances across different refactorings - $scenario_B$. . 104
4.14 CPS antipattern instances across different refactorings - $scenario_A$. . . 105
4.15 P&F antipattern instances across different refactorings - $scenario_C$. . . 106

5.1 Java method `minimumPathSum` for calculating the minimum path sum in a $M \times N$ matrix. 113
5.2 Overview of the integrated approach for software performance engineering 115
5.3 DTMC model for `minimumPathSum` Java method. 119
5.4 PRISM model synthesised for the `minimumPathSum` Java method. . 120
5.5 Java implementation of the `search` method 122
5.6 DTMC model representation of the `search` Java method 123
5.7 BLOB antipattern instances while varying operational profiles. 124
5.8 BLOB antipattern instances when improving the rate of the TA component (code-level refactoring). 124
5.9 BLOB antipattern instances when introducing a second thread of the TA component (architecture-level refactoring). 125
5.10 Java implementation of the `search` method after code restructuring has been applied 127
5.11 Updated DTMC model representation of the `search` Java method . . 128

A2.1 Java method `DevPerf` from the Telegram android application 145
A2.2 PRISM model synthesised for the `DevPerf` Java method 146
A3.1 Java method `fst` from the Apache Commons Math library 148
A3.2 PRISM model synthesised for the `fst` Java method 149
A4.1 Java implementation of the `knapsack` algorithm 150
A4.2 PRISM model synthesised for the `knapsack` Java implementation . 151

B1.1 DTMC model for MW synthesised using the PAR pattern 154
B1.2 DTMC model for FA synthesised using the SEQ pattern 155
B1.3 DTMC model for Order synthesised using the PROB pattern 156
B2.1 CTMC model representation of the FX system consisted of two threads and two instances of TA component 159
B3.1 Number of Order service invocations per time unit 160
B3.2 The path probability of executing the FA and Order external services, and the Notification internal component 161
B3.3 TA internal component's response time 161

List of Tables

2.1 System requirements for the weather prediction DTMC model 42
2.2 System requirements for the queue of jobs CTMC model 44
2.3 Summary of probabilistic model checking tools 46
2.4 Overview of performance antipatterns, adopted from [158] 48

3.1 Description of case studies' models and properties of interest expressed in both natural language and PCTL. 70
3.2 Comparison in accuracy of results obtained using PROPER and simulation. 71
3.3 Time and memory consumption comparison between PROPER and simulation. 72
3.4 Results obtained using PROPER for two different scenarios. Scenario A: replacement of a program method with a functionally-equivalent method with different performance characteristics. Scenario B: Program deployment on a new hardware platform with different quality attributes. 73
3.5 Overview of related research 77

4.1 Detection rule parameters. 92
4.2 System parameters. 98

To my parents Panagiota and Giorgos.
Thank you for always believing in me.

Acknowledgements

First, I would like to express my most sincere gratitude and appreciation to my supervisors Prof. Radu Calinescu and Dr. Simos Gerasimou for their constant and indispensable support throughout the duration of this research project. Their guidance, expertise and patience have been vital for the materialisation of this thesis.

Special thanks to Prof. Ana Cavalcanti, my internal assessor, for her invaluable feedback and continuous encouragement, and Prof. Vincenzo Grassi for his suggestions to improve this thesis.

I would like to thank Microsoft Research for supporting this project through its PhD Scholarship Programme, and for giving me the opportunity to attend the thought-provoking seminars of their PhD Summer School and interact with fellow early career researchers.

I would also like to thank all the members of both the Trustworthy Adaptive and Autonomous Systems Processes and the Automated Software Engineering research groups for all the interesting and engaging discussions. I am especially grateful to Dr. Colin Paterson for his helpful advice and ideas. Special thanks also goes to my colleagues Misael Alpizar Santana, Saud Yonbawi, Naif Alasmari and Emad Alharbi with whom I shared a big part of this journey, and to Dr. Calum Imrie, Gricel Vazquez Flores, Joshua Riley, and Dr. Xinwei Fang for their feedback during a mock viva.

During my study I had the opportunity to collaborate with great researchers and academics. A special thank you goes to Prof. Vittorio Cortellessa and Dr. Catia Trubiani.

Finally, I wish to express my deepest gratitude to my family for their endless support, encouragement and patience; without their love, nothing would be possible.

Declaration

I hereby declare that all of the work contained in this thesis, except where specific reference is made to the work of others, represents the original contribution of the author. This work has not previously been submitted for consideration for an award at this, or any other university. All sources are acknowledged as References.

Chapter 4 describes joint work with Prof. Vittorio Cortellessa from University of L'Aquila, Italy and Dr. Catia Trubiani from Gran Sasso Science Institute, Italy. We clarify our contributions for this chapter in Section 1.2.

Parts of the research described in this thesis have been previously published in:

- Ioannis Stefanakos. **Towards Integrated Correctness Analysis and Performance Evaluation of Software Systems (Doctoral Forum Paper)**. In 11th International Workshop on Software Engineering for Resilient Systems (SERENE), pages 109–117, 2019.
- Radu Calinescu, Vittorio Cortellessa, Ioannis Stefanakos and Catia Trubiani. **Analysis and Refactoring of Software Systems Using Performance Antipattern Profiles**. In 23rd International Conference on Fundamental Approaches to Software Engineering (FASE), pages 357–377, 2020. (NB: The author list for this work is ordered alphabetically.)
- Ioannis Stefanakos, Radu Calinescu and Simos Gerasimou. **Probabilistic Program Performance Analysis**. In 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pages 148–157, 2021. (Best paper award candidate)
- Ioannis Stefanakos, Simos Gerasimou and Radu Calinescu. **Software Performance Engineering with Performance Antipatterns and Code-level Probabilistic Analysis**. In 18th Workshop on Model Driven Engineering, Verification and Validation, 21st International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pages 249-253, 2021.

Ioannis Stefanakos

December 2021

Chapter 1

Introduction

> Whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen.
>
> P. Clements and L. Northrop
> Software Engineering Institute
> Carnegie Mellon University, 1996

It has long been known that software systems exhibit errors. In order to increase the dependability of these systems, software engineers devote a substantial amount of time to their testing and debugging. There has always been research focusing on developing new or improving the existing software verification methods [63]. Verification is the area that includes all the techniques aiming to improve software quality—and to provide evidence that the final product conforms to the specified requirements [89]. Some of the techniques subsumed under verification are [115] formal verification [57, 97, 135], testing [13, 19, 150] and simulation [23, 90, 154].

Testing is considered an essential activity in software engineering [121]. It is defined as the process of examining the system's behaviour with the aim of identifying potential malfunctions [20]. With the increase of involvement of software and hardware systems in our everyday lives, testing has become more complex but at the same time necessary to ensure the correct functionality of these systems. Although successful testing identifies a significant amount of errors, it is still impossible to capture all of them [112], especially in dynamic environments. As a result, computer systems

undergoing only testing can still be afflicted by errors which could potentially lead to severe consequences, e.g., in the case of safety-critical systems.

The need for guaranteeing the correct operation of computer systems led to the development of formal verification techniques such as model checking [40] and stochastic verification [104] (probabilistic model checking) which are able to address issues that testing alone is not able to identify. Both of these techniques are also known as formal methods, which is a line of study that takes advantage of the fact that that computer systems can be depicted as mathematical objects whose behaviour is, in principle, well determined [63]. Model checking establishes mathematical guarantees that the functional requirements of a system are met, and in case of requirement violations, counterexamples are provided as a proof of failure. While model checking has become a common activity during hardware and software design, it provides absolute guarantees of system correctness (e.g., the system's processes will never fail) that may not be applicable to systems that exhibit stochastic behaviour or operate in uncertain environments. Software systems of this nature can be found in applications used in aircrafts and vehicles, as well as in personal devices such as mobile phones. In order to be able to verify the correctness of these systems, it is necessary to analyse quantitative properties such as performance and reliability. Probabilistic model checking can establish such properties by providing a probabilistic bound on the result, and thus probabilistic guarantees such as 'the probability of failure for any of the system's processes is less than 0.05%'.

While the functionality delivered by a software system is evidently important, it should not be the only focus of verification during the software development process. Over the last two decades, research has shown the significance of integrating activities for nonfunctional property analysis into the development process, with the aim of meeting nonfunctional requirements. As stated in [148], "*the cost of a software product over its lifetime is determined more by how well it achieves its objectives for quality attributes such as performance, reliability or maintainability, than by its functionality*". The quality of the final software product can be vastly affected during the early stages of the development process, and wrong decisions early on may require costly modifications late in the development process, even involving changes to the overall design of the system. Therefore, the system's quality attributes must be assessed early in the software development process to avoid the violation of requirements and failure of entire projects. Consider the following example quoted from [145]:

> *The National Aeronautics and Space Administration (NASA) was forced to delay the launch of a satellite for at least eight months. The satellite and*

> *the Flight Operations Segment (FOS) software running it are a key component of the multi billion-dollar Earth Science Enterprise, an international research effort to study the interdependence of the Earth's ecosystems. The delay was caused because the FOS software had unacceptable response times for developing satellite schedules, and poor performance in analyzing satellite status and telemetry data. There were also problems with the implementation of a control language used to automate operations. The cost of this rework and the resulting delay has not yet been determined. Nevertheless it is clearly significant, and the high visibility and bad press is potentially damaging to the overall mission. Members of Congress also questioned NASA's ability to manage the program.*

The success of today's software systems is critically affected by their performance. However, a large percentage of the produced software fails to meet the performance objectives set during the design stage. As there are many inter-dependencies and trade-offs between quality attributes within a system, fixing occurring problems at later stages can be both costly and challenging.

1.1 Motivation and Research Hypothesis

Software is among the most flexible engineering artifacts. Computer code can run unmodified on hardware platforms as different as desktop PCs and smartphones, or with different *usage profiles* (i.e., probability distributions of the program inputs). Even when the code is modified, the change can be localised: a function or module is easy to replace with a functionally equivalent one that is, for instance, faster or more reliable.

This flexibility is a great strength, but makes the analysis of the performance and other quality aspects of software systems very challenging. Changes in platform, usage profile and individual functions or modules may not affect the functionality of programs, but can impact their execution time and use of resources significantly. Given the importance of these properties, software performance analysis has been the subject of intense research for several decades [5, 12, 102, 131]. Nevertheless, the solutions delivered by this research focus on analysing the performance of software at architectural level, e.g. [16, 45, 49, 71, 79].

The equally important and challenging analysis of software performance at code-level is typically carried out through program instrumentation, monitoring and profiling [4, 11, 103, 139, 163]. While these techniques produce accurate results, they have

the significant drawback that the code needs to be actually executed for every platform and usage profile of interest, and after every code change.

Performance antipatterns [24, 149] and stochastic modelling (e.g., using queueing networks, stochastic Petri nets, and Markov models [21, 50, 157]) have long been used in conjunction, to analyse performance of software systems and to drive system refactoring when requirements are violated. End-to-end approaches supporting this analysis and refinement processes have been developed (e.g., [6, 26, 74]), often using established tools for the simulation or formal verification of stochastic models of the software system under development (SUD).

While these approaches can significantly speed up the development of systems that meet their performance requirements, they are only applicable when the SUD operational profile (i.e., a quantitative characterization of the system's intended behaviour [119]) is known and does not change over time. Both of these are strong assumptions. In practice, software systems are often used in applications affected by uncertainty, due both to incomplete knowledge of and to changes in workloads, availability of shared resources, etc.

This thesis presents research that complements, and addresses several limitations of, the solutions summarised above. To that end, we introduce a new end-to-end software performance methodology that enables the performance analysis of software systems at both code-level and system architecture-level. At code-level, we synthesise a discrete-time Markov model representations of the code of interest, and we analyse the nonfunctional properties for this code. At system architecture-level, we combine in a new way probabilistic modelling and performance antipatterns. Finally, we integrate the two methods into a novel end-to-end software performance engineering approach. The hypothesis underlying the research in this thesis is as follows:

> *Given the formalisation of the architecture of a software system and/or of the source code of its relevant components as discrete-time Markov chains, and a set of nonfunctional requirements encoded in probabilistic temporal logic, probabilistic model checking combined with performance antipatterns can:*
>
> 1. *provide guarantees that these requirements are met for certain operational-profile regions;*
> 2. *guide the refactoring of the software system to ensure it meets the requirements for operational-profile regions of interest for which its initial version violates these requirements.*

1.2 Contributions

The main contributions of the thesis, described in Chapters 3–5, are summarised below.

Probabilistic Analysis of Code Performance

In Chapter 3 we introduce a tool-supported method (PROPER) for the formal analysis of timing, resource use, cost and other quality aspects of computer programs. PROPER (a) synthesises a Markov-chain model of the analysed code, (b) computes this quantitative model's transition probabilities using information from program logs, and (c) employs probabilistic model checking to evaluate the performance properties of interest. Unlike existing solutions, our method can reuse the probabilistic model to accurately predict how the program performance would change if the code ran on a different hardware platform, used a new function library, or had a different usage profile.

We evaluate PROPER through its application in analysing the performance of Java code that is used in real-world systems. Specifically, the fragments of Java code used in the PROPER evaluation from Chapter 3 come from the Apache Commons Math library, the Android messaging app Telegram, and an existing implementation of the Knapsack algorithm. We demonstrate the method's effectiveness by proving its (1) accuracy, i.e., the supported analysis of nonfunctional properties of interest can achieve the same accuracy compared to other established alternatives, (2) decision-making, i.e., our method is able to correctly guide software engineers in their decision-making, and (3) efficiency, i.e., our method has lower computational overheads than alternative approaches.

Software System Analysis & Refinement Using Performance Antipattern Profiles

In Chapter 4 we propose a novel method that uses performance antipatterns and stochastic modelling to support the analysis and refinement of software systems at architectural level. The new approach computes the performance antipatterns present across the operational profile space of a software system under development, enabling engineers to identify operational profiles likely to be problematic for the analysed design, and supporting the selection of refactoring actions when performance requirements are violated for an operational profile region of interest.

We demonstrate the applicability of our approach for a software system comprising a combination of internal (i.e., in-house) components and external third-party services. The chosen system, adapted from [34], is based on a real-world service-based system from the foreign exchange trading domain. More information on the system can

be found in Section 4.1. Our experiments show that the approach can successfully provide insights on the performance antipattern profile of a specific design, and support performance-driven refactoring decisions on the basis of the performance antipattern profile.

Since this research involved collaborative work, it is important to clarify that our collaborators provided performance antipattern expertise, supervisory support (particularly on the aspects of the work requiring this expertise), and input into the planning and evaluation of the experimental results. In turn, the PhD project involved the conceptualisation and development of the proposed approach, the development of the case study and its use to evaluate the approach (including the stochastic modelling, requirement specification, and the execution of all the experiments), and the development of the tool that automates several steps of the approach.

Software Performance Engineering With Code-level Probabilistic Analysis and Performance Antipatterns

In Chapter 5 we introduce a new approach that integrates the two software performance engineering methods summarised above. The integrated approach obtained in this way allows software engineers to verify performance properties at both code-level and system architecture-level, supporting the selection of refactoring actions when a performance requirement violation occurs. Monitoring the effect of changes in the system's code and architecture gives a better overview of (and control over) the system's performance objectives.

To validate the integrated methodology, we assess its application in a case study involving the performance analysis and refactoring of the foreign exchange trading service-based system introduced in Chapter 4, which comprises a combination of internal components and third-party services. The results of this evaluation show the feasibility of meaningfully using results obtained via PROPER for software performance engineering at system architecture-level, enabling a combined performance analysis approach.

1.3 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 contains the background information, essential for the development of our techniques in later chapters. Section 2.1 briefly presents model checking, the need for such technique, and some of its main characteristics for the purpose of introducing

probabilistic model checking in Section 2.2 and to enable us to dive deeply into the semantics of this follow-up technique. Sections 2.2.1 and 2.2.2 introduce the Markov chain models and probabilistic variants of temporal logic used throughout the thesis, respectively. Section 2.2.3 overviews the available tools that automate the verification techniques employed by probabilistic model checking. Section 2.3 presents the concept of performance antipatterns, focusing on a more detailed description and examples of the three performance antipatterns exploited in the thesis. Finally, Section 2.4 provides a summary of the chapter.

Chapter 3 describes PROPER, our probabilistic analysis of code performance method. PROPER synthesises discrete-time Markov chain (DTMC) model representations of Java source code, and uses probabilistic model checking to verify nonfunctional properties of interest. The chapter starts with an introduction of the main aim and the contributions of this work. Section 3.1 presents a motivating example that showcases PROPER's application. Sections 3.2.1–3.2.4 provide a detailed analysis of the method by breaking down its steps, and give insight into how PROPER can be used in further application scenarios. Specifically, the probabilistic model synthesis step is detailed in Section 3.2.1, followed by the calculation of transition probabilities (Section 3.2.2), the application of probabilistic model checking (Section 3.2.3) and further application scenarios for PROPER (Section 3.2.4). Section 3.3 describes our open-source PROPER tool that automates the probabilistic model synthesis step and enables the generation of DTMCs from input Java code. Sections 3.4.1–3.4.4 present PROPER's evaluation by highlighting our research questions (Section 3.4.1), explaining our experimental setup (Section 3.4.2), discussing the obtained results (Section 3.4.3) and presenting the identified threats to validity (Section 3.4.4). Sections 3.5 and 3.6 compare PROPER with existing approaches and summarise the chapter, respectively.

Chapter 4 introduces our software system analysis and refinement approach that makes use of performance antipattern profiles. The introductory paragraphs of the chapter present the approach, listing the key points it tries to address and its main contributions to the area of software systems' architecture analysis and refinement. Sections 4.1.1–4.1.4 introduce a motivating example that showcases the benefit of applying our approach in a case study from the foreign currency exchange domain. Sections 4.2.1–4.2.5 describe the steps of the approach. We start with the modelling step (Section 4.2.1), where the synthesis of parametric Markov models takes place, followed by the model instantiation (Section 4.2.2) and analysis (Section 4.2.3), and the generation of antipattern profiles for the analysed system (Section 4.2.4). The description of the approach ends with the refactoring step (Section 4.2.5), which assesses whether any refactoring actions are required. Section 4.3 describes our implemented

tool for the automatic generation of antipattern profiles. Sections 4.4.1–4.4.4 present the evaluation of the approach by first defining the research questions that will be explored (Section 4.4.1, followed by the experimental scenarios (Section 4.4.2) and a discussion on the obtained results (Section 4.4.3), and concludes with presenting the identified threats to the validity of the approach (Section 4.4.4). Sections 4.5 and 4.6 compare our software system analysis and refinement approach with existing approaches and summarise the chapter, respectively.

Chapter 5 describes the combination of the two methods from Chapters 3 and 4 into an end-to-end methodology that enables the performance analysis of software systems at both code-level and system architecture-level. The introduction of this chapter presents the new combined approach and highlights the aim of this integration. Section 5.1 illustrates the application of the approach using a motivating example that combines the FX system, first presented in Section 4.1, and a Java representation of the minimum path sum algorithm. Sections 5.2.1–5.2.5 describe the steps of the integrated approach, starting with the identification of the system components of interest (Section 5.2.1). Then, PROPER is employed to generate the DTMC model representations based on the source code of these components (Section 5.2.2), and its results are given as input to the architecture-level model synthesised in the next step (Section 5.2.3). Following the model construction, the antipattern-based analysis method is used to analyse the system model and check whether requirements are violated (Section 5.2.4). The last step involves the initiation of refactoring actions if required and the option of modifying both the system's internal components and the way in which it uses external services (Section 5.2.5). Section 5.3 evaluates the approach, and Section 5.3.4 discusses threats to validity. Sections 5.4 and 5.5 compare our integrated software performance engineering methodology with existing approaches and conclude the chapter, respectively.

Chapter 6 concludes the thesis by summarising the findings and contributions of this research, and discusses directions for future work.

Chapter 2

Background

Our reliance on software and software-controlled systems is rapidly growing, as does the complexity of these systems. Embedded systems such as smart cards, personal computers and mobile phones, and online services including banking and healthcare are now an integral part of daily life.

Traditional verification techniques, such as testing and simulation, are often not effective in finding errors in systems of high complexity or when these systems exhibit probabilistic behaviour, e.g., due to operating in uncertain environments. To address the limitations of these techniques, formal verification has been introduced. Formal verification is capable of providing correctness *guarantees* for certain properties of a software system. Formal verification techniques have become prominent over the years for ensuring not only the correct functionality of systems, but also for establishing their quality attributes such as performance and reliability.

This chapter introduces terminology and concepts used throughout the rest of the thesis. Specifically, we focus on modelling and analysis techniques that allow for the formal verification of probabilistic models, and for identifying bad software engineering practices which can lead to system performance degradation. These tehcniques underpin the work undertaken later in the thesis. Section 2.1 introduces the concept of model checking. Section 2.2 then defines probabilistic model checking, the two model variants used in this thesis (i.e., discrete-time and continuous-time Markov chain models) and the temporal logics necessary for analysing the properties of these probabilistic models, and briefly outlines the use of probabilistic model checking tools. Section 2.3 defines performance antipatterns, with a focus on the antipatterns used later in the thesis to assess the performance of software systems. Finally, Section 2.4 provides a summary of the chapter.

2.1 Model Checking

Model checking is an automated technique that focuses on the verification of finite-state systems. The technique was firstly developed by Clarke and Emerson in 1981 [40] and independently discovered by Queille and Sifakis in 1982 [43]. The system semantics are given by means of a Kripke structure, and the specification of its properties is expressed using temporal logic [41]. Clarke and Emerson mentioned that the use of the name *Model Checking* resulted from their work in determining whether a Kripke structure M was a model for a formula of temporal logic f [40]. In model checking, both the system and the specification must be formally described; that is why the use of Kripke structures and temporal logic is necessary.

A Kripke structure is a labeled graph that contains all the possible states of a system and the transitions between them. The states are represented by the vertices of the graph and the transitions by the graph's edges. Formally, a Kripke structure is a tuple $M = (S, s_0, \mathbf{R}, AP, L)$, where S is a set of states, $s_0 \subseteq S$ is a set of initial states, $\mathbf{R} \subseteq S \times S$ is a transition relation, AP is a set of atomic propositions and $L : S \to 2^{AP}$ is a labeling function that maps each state to the set of propositional variables from AP that hold in it.

Figure 2.1 shows a simple four-state Kripke structure model of an alarm clock, adapted from [43]. The four states (s_0, s_1, s_2 and s_3) correspond to the alarm being *off*, *on*, *ringing* and *snoozed*, respectively. One of the system's transitions is from state s_2 to state s_3, meaning that it is possible for the alarm clock to go from its ringing state to the state where it has been snoozed. However, there is no transition from state s_1 to state s_3, as the alarm cannot be snoozed if it is not ringing.

Temporal logic, which is an extension of propositional logic [100], is a "*formalism for reasoning about time without introducing it explicitly*" [43]. There are two main views regarding the nature of time. One considers time to be linear and is called Linear-Time Temporal Logic (LTL), and the other that time has a branching, tree-like nature, and is called Computation Tree Logic (CTL), i.e., LTL takes into account one path at a time, while CTL can consider multiple paths. The syntax of both LTL and CTL consists of logical and temporal operators, e.g., $\neg, \vee, G$ for always, F for eventual. Another type of temporal logic CTL* has been introduced in [64], and supports the expressions of both CTL and LTL. The reason behind the creation of CTL* was to address issues that result from nondeterminism that is present in many concurrent programs. All these types of temporal logic can be used to express properties verified by model checking.

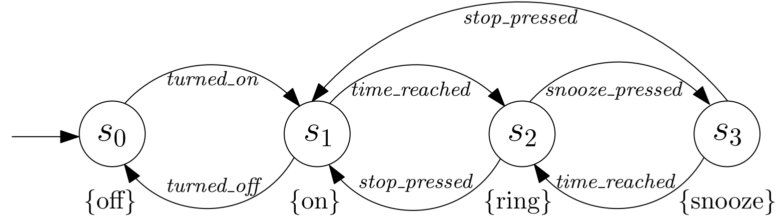


Fig. 2.1: State transition system of an alarm clock, adapted from [43]

Consider again the *alarm* model in Figure 2.1. The initial state satisfies the formula $\mathbf{AG}\neg(\mathit{ring} \wedge \mathit{snooze})$, where **A** is the universal path quantifier and **G** (globally) is a temporal operator, since no state exists that is both *ringing* and *snoozing*. On the other hand, it does not satisfy the formula $\mathbf{E}[\neg \mathit{ring}\, \mathbf{U}\, \mathit{snooze}]$, where **E** is the existential path quantifier and **U** (until) is a temporal operator, since in order to reach a *snoozed* state, a *ringing* state has to be reached first.

Model checking is used to provide absolute guarantees of correctness for software systems, such as “deadlock freeness” or “the program eventually terminates”. However, many real-world systems that operate in uncertain environments exhibit stochastic behaviour and, thus, rigid guarantees cannot be established. Probabilistic model checking (also known as quantitative verification) is a formal verification technique suitable for this purpose. It uses finite state transition models extended with probabilities/rates and verifies properties expressed in probabilistic temporal logic.

2.2 Probabilistic Model Checking

Probabilistic Model Checking (PMC) [95, 104] is a formal verification technique used to establish the correctness, reliability and performance of systems with stochastic behaviour, where this behaviour is formalised using Markov models (see Section 2.2.1). These models are Kripke structures whose edges are annotated with the probability or rate of taking the associated transition, depending on the model type. To enable the analysis of additional types of properties, Markov models can be augmented with cost/reward structures that associate non-negative values with their states and transitions.

The properties of interest analysed using these models are formally expressed in probabilistic variants of temporal logic (see Section 2.2.2). These expressions (i.e., specifications) enable reasoning about the probability of different events occurrencing during the operation of the modelled system, or about the costs/rewards associated with these events. The specified properties capture not only the system’s “correctness”, but also a variety of its quantitative characteristics, such as reliability and performance.

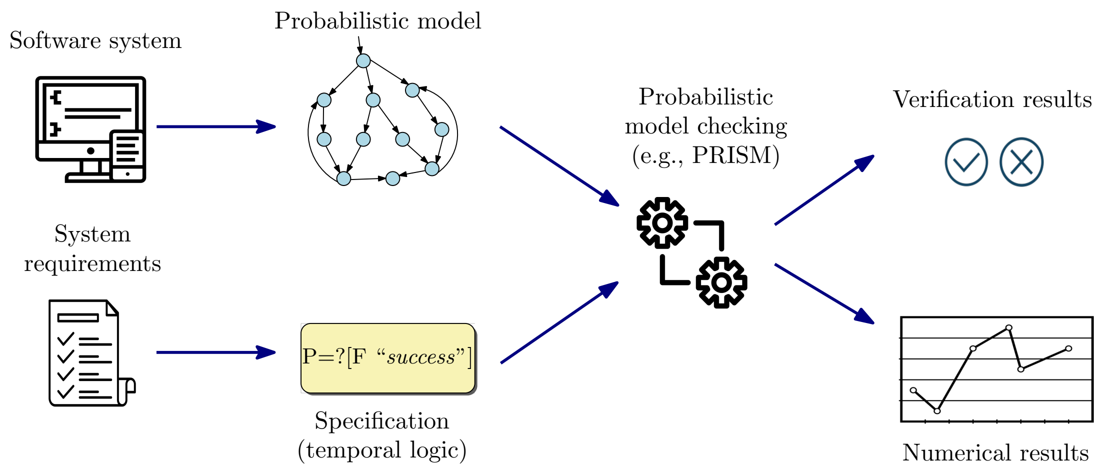


Fig. 2.2: Overview of probabilistic model checking, adapted from [123]

Examples of properties expressed in both natural language and probabilistic temporal logic are the following:

- For a system in which failures may occur:

  "What is the probability of successful program termination?" : $P =?[F \text{ ``}success\text{''}]$

- For a battery-powered device:

  "What is the expected power consumption?" : $R\{\text{``}energy\text{''}\} =?[F \text{ ``}success\text{''}]$

where F (eventually) is a temporal operator and *"success"* is a label referring to the model's final state, i.e, successful termination.

An overview of PMC adapted from [123] is depicted in Figure 2.2. A Markov model representing the system under consideration and a list of properties expressed in probabilistic temporal logic are given as input to a probabilistic model checker. An exhaustive analysis is then performed to establish the values of the properties and produce the results of verification. By using PMC, the engineers can determine if a given property meets a specified bound (or a given threshold) and thus be informed about the satisfaction or not of the system's requirements. Moreover, verification results in the form of probabilities or costs/rewards associated with a particular event can be used to examine the system's behaviour and support decision making. To enable the practical application and automation of PMC, probabilistic model checking tools such as PRISM [109], MRMC [96] and Storm [60] have been implemented and widely used both by the research community and industry. A discussion about probabilistic model checkers is given later in this chapter (see Section 2.2.3).

The increasing role of probability in the design and analysis of software systems, and the importance of techniques such as PMC, can be seen in their practical applications [134]. Nowadays, it is common for many systems to be deployed in dynamic or uncertain environments. This stochastic behaviour can be modelled and predicted using PMC. Additionally, the randomisation in distributed algorithms can be used as a symmetry breaker and vastly contribute in devising efficient solutions; e.g., randomised leader election [91] assumes a random identity from a finite domain for each of the available processors, which they send around the network. Following the algorithm's successful termination, a leader is selected (with high probability). Finally, probability can be used to model system failures and the overall system performance. Typical reliability and performance examples in a service-based system include, "the probability that the system fails within 60s is at most 0.001" and "the expected response time of service X must not exceed 10ms", respectively.

The remainder of this section is split into three parts. The definition of the Markov chain model variants used by the methods developed in this thesis is presented in Section 2.2.1 and covers both discrete-time and continuous-time domains. The definition of probabilistic temporal logics, necessary for quantifying system properties using the synthesised Markov chain models is provided in Section 2.2.2. Finally, Section 2.2.3 presents widely used probabilistic model checking tools that automate the verification of Markov models.

2.2.1 Markov Models

Markov models are commonly used to represent software systems that exhibit probabilistic behaviour, and to make predictions about them. Specific aspects of a system's behaviour can then be modelled as a stochastic process, which is described as a system evolving in time while undergoing chance fluctuations [44]. Formally, a stochastic process is defined as a collection of random variables $\{X(\tau)\}$, indexed by a parameter τ, where τ is interpreted as time and belongs to an index set T, and $X(\tau)$ is the state of the process at time $\tau \in T$ [132]. Additionally, the set S of all possible $X(\tau)$ values is called the *state space* of the stochastic process. If T is a set of integers, i.e., $T \subseteq \mathbb{Z}^+ = \{0, 1, 2, \dots\}$, representing specific time points, we have a *discrete-time* process $\{X(\tau)\}$; otherwise, if T is the real line (or some interval of the real line), i.e., $T \subseteq \mathbb{R}^+ = [0, \infty)$, we have a *continuous-time* process $\{X(\tau)\}$.

A stochastic process can be also characterised by the fact that it retains no memory of previous states, i.e., only the current state can influence its evolution [124]. Such a process that satisfies the Markov property (Def. 2.1), also known as *memoryless*

property, is called a Markov process. The term *memoryless* property results from the fact that a state of the stochastic process at time-step τ is decided only by the state at the previous time-step $\tau - 1$.

Definition 2.1. *A stochastic process* $\{X(\tau)\}$ *satisfies the Markov property if*

$$P\{X(\tau) = s_\tau \mid X(\tau-1) = s_{\tau-1}, \ldots, X(0) = s_0\} = P\{X(\tau) = s_\tau \mid X(\tau-1) = s_{\tau-1}\} \tag{2.1}$$

where $s_0, \ldots, s_n \in S$ *represent successive states of the process and* $\tau > 0$*.*

All models defined in the following sections are variants of the Markov process. While there are many types of probabilistic models [95], our focus is on discrete-time and continuous-time Markov chains.

2.2.1.1 Discrete-Time Markov Chains

Discrete-Time Markov chains (DTMCs) are labelled state-transition systems augmented with probabilities. A DTMC consists of a discrete set of states representing possible configurations of the system being modelled. The transitions between states occur in discrete time-steps, and the probability of making these transitions between states is given by discrete probability distributions. The following formal definition was adapted from [104].

Definition 2.2. *A discrete-time Markov chain (DTMC) is a tuple* $D = (S, s_0, \boldsymbol{P}, AP, L)$ *where:*

- S *is a finite set of states;*
- $s_0 \in S$ *is the initial state;*
- $\boldsymbol{P} : S \times S \rightarrow [0,1]$ *is a transition probability matrix, such that for all* $s \in S$*:*

$$\sum_{s' \in S} \boldsymbol{P}(s, s') = 1; \tag{2.2}$$

- AP *is a set of atomic propositions (i.e., statements associated with system states and that evaluate to true or false depending on whether they hold or not);*
- $L : S \rightarrow 2^{AP}$ *is a state labelling function that maps each state* $s \in S$ *to the set* $L(s) \subseteq AP$ *of atomic propositions that hold in state* s*.*

Every element $\mathbf{P}(s, s')$ of the transition probability matrix $\mathbf{P}$ gives the probability of taking a transition from state s to state s'. The transition probabilities associated with the outgoing transitions from a single state must sum up to one. States with only an

outgoing transition to themselves are called *terminating states*, and are modelled by adding a self loop with probability of 1 [105]. Additionally, the behaviour of a DTMC model is represented as a set of paths. A *path* π is a non-empty sequence of states $\pi = s_0, s_1, s_2, \ldots$ where $s_i \in S$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. Paths can either be finite or infinite. The i-th state of a path π is denoted by $\pi(i)$ and its length by $|\pi|$. The set of all paths of a DTMC D, starting in a given state s, are denoted as $Path^D(s)$.

To reason about the stochastic behaviour of the DTMC, we need to obtain the probability that certain paths are taken. This is accomplished by defining, for each state $s \in S$, a probability measure Pr_s over the set of paths $Path^D(s)$. The probability measure is then induced by the probability matrix $\mathbf{P}$ as follows:

Definition 2.3. *For any finite path $\pi = s_0, s_1, s_2, \ldots, s_n \in Path^D(s)$:*

$$\boldsymbol{P}_s(\pi) = \begin{cases} 1 & if\ n = 0 \\ \boldsymbol{P}(s_0, s_1) \cdot \boldsymbol{P}(s_1, s_2) \cdot \ldots \cdot \boldsymbol{P}(s_{n-1}, s_n) & otherwise \end{cases} \tag{2.3}$$

where s_0 is the starting state and s_n is the target state.

More details can be found in [134, 9].

Extending DTMCs with rewards

The types of properties that can be analysed using DTMCs can be enhanced through the addition of cost/reward structures. These refer to non-negative, real-valued quantities that are associated with states and transitions in the model. The difference between a cost and a reward is purely semantic. There is no mathematical distinction, just a commonly adopted notion that costs should be minimised and rewards should be maximised.

For a DTMC $D = (S, s_0, \mathbf{P}, AP, L)$ a reward structure $(\underline{\rho}, \iota)$ enables the specification of two distinct types of rewards: *state* rewards and *transition* rewards. The former are assigned to states by means of the reward function $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$, and the latter, are assigned to transitions by means of the reward function $\iota : S \times S \to \mathbb{R}_{\geq 0}$. The state reward $\underline{\rho}(s)$ is obtained in state s per occurred time-step, and the transition reward $\iota(s, s')$ is obtained each time a transition between states s and s' takes place. State and transition rewards are also known as cumulative and instantaneous rewards, respectively.

Definition 2.4. *A cost/reward structure over a DTMC $D = (S, s_0, \boldsymbol{P}, AP, L)$ is a pair of real-valued functions $(\underline{\rho}, \iota)$ where:*

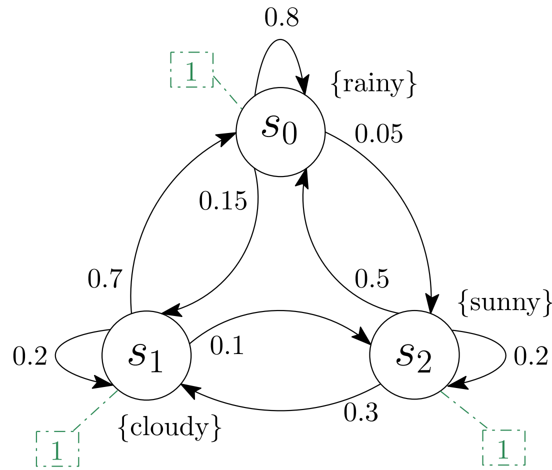


Fig. 2.3: DTMC weather prediction model, adapted from [154]

- $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ *is a state reward function that defines the value (cost/reward)* $\rho(s)$ *obtained when D is in state* $s \in S$ *for one time step.*
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ *is a transition reward function that defines the value (cost/reward)* $\iota(s, s')$ *obtained each time a transition occurs from state* $s \in S$ *and to state* $s' \in S$.

Example 2.1. *Figure 2.3 depicts a DTMC that describes an imagined daily weather pattern for York, UK (well known for its unpredictable weather). The weather model is simplified by considering only three types of weather: rainy, cloudy and sunny. The three weather conditions are associated with the three states of the DTMC. State* s_0 *represents a rainy day; state* s_1 *a cloudy day; and state* s_2 *a sunny day. The weather is observed daily, and on any given rainy day, the probability of rainfall on the next day is estimated at 0.8, and the probabilities that the next day will be either cloudy or sunny are 0.15 and 0.05, respectively. Probabilities are also assigned when a particular day is cloudy or sunny as shown in the weather model's depiction. Additionally, the weather model is augmented with a state reward structure, shown in green colour, that associates a time of 1 second with each observation (i.e., it takes 1 second to report the results when an observation state is reached).*

The elements of the DTMC model are:

The set of states $S = \{s_0, s_1, s_2\}$

The initial state s_0

The transition probability matrix $\boldsymbol{P} = \begin{pmatrix} 0.8 & 0.15 & 0.05 \\ 0.7 & 0.2 & 0.1 \\ 0.5 & 0.3 & 0.2 \end{pmatrix}$

The set of atomic propositions $AP = \{rainy, cloudy, sunny\}$

The labelling function L : $L(s_0) = \{rainy\}$, $L(s_1) = \{cloudy\}$, $L(s_2) = \{sunny\}$

*All elements in **P** represent conditional probabilities. For example, given that today is sunny, suppose we want to estimate the probability that tomorrow is cloudy and the day after is rainy, i.e., the probability for the path* $\pi = s_2, s_1, s_0$*:*

$$\boldsymbol{P}_s(\pi) = \boldsymbol{P}(s_2, s_1) \cdot \boldsymbol{P}(s_1, s_0) = 0.3 \times 0.7 = 0.21$$

Each state (s_0, s_1, s_2) *within the weather model is associated with a reward. The rewards correspond to the time needed to complete the observation process and report back the result. The value of each reward is equal to 1 and their structure is defined as:*

$$\iota = 0_{3,3}$$
$$\underline{\rho}^{time} = (1,1,1)$$

where $0_{3,3}$ *denotes a* 3×3 *matrix populated with zeros, and (1,1,1) refers to the reward value in each of the existing states.*

2.2.1.2 Continuous-Time Markov Chains

Continuous-Time Markov Chains (CTMCs) allow the modelling of systems consisting of discrete states, but where time progresses continuously. The transitions between states occur at any point in time, and the delays are modelled by exponential probability distributions [108]. Like DTMCs, CTMCs satisfy the Markov (memoryless) property, i.e., the probability of transitioning to another state depends only on the current state. Additionally, the probability is independent of the amount of time spent in the current state. The application of CTMCs can be observed across various fields, where they are used to analyse nonfunctional properties of systems such as performance and reliability. Typical examples of such systems are control systems, queuing networks, and biological pathways.

Definition 2.5. *A continuous-time Markov Chain (CTMC) is a tuple* $C = (S, s_0, \boldsymbol{R}, AP, L)$ *where:*

- *S is a finite set of states;*
- $s_0 \in S$ *is the initial state;*
- $\boldsymbol{R} : S \times S \rightarrow \mathbb{R}_{\geq 0}$ *is a transition rate matrix;*
- *AP is a set of atomic propositions;*

- *$L : S \to 2^{AP}$ is a state labelling function that maps each state $s \in S$ to the set $L(s) \subseteq AP$ of atomic propositions that hold in that state.*

The transition rate matrix $\mathbf{R}$ is responsible for associating rates with each pair of states in the CTMC. A transition between any states s and $s' \in S$ can only occur if $\mathbf{R}(s,s') > 0$, and the probability of a transition within t time-units is given by the negative exponential distribution $1 - e^{-\mathbf{R}(s,s')\cdot t}$. Given a state s, if there is more than one state s' such that $\mathbf{R}(s,s') > 0$, the first transition to occur determines the next state of the CTMC (race condition). The time spent in state s, before a transition occurs, is exponentially distributed with rate:

$$E(s) = \sum_{s' \in S} \mathbf{R}(s,s') \tag{2.4}$$

where $E(s)$ is the *exit rate* of state s. Unlike DTMCs, states in CTMCs might have no outgoing transitions, and such a state is called an *absorbing state*. It is also possible to determine the probability of moving to state s' when leaving state s, independently of the time spent in s. This is achieved by extracting the *embedded* DTMC from the CTMC.

Definition 2.6. *The embedded DTMC D of a CTMC $C = (S, s_0, \boldsymbol{R}, AP, L)$ is a tuple $emb(C) = (S, s_0, \boldsymbol{P}^{emb(C)}, AP, L)$ where for $s, s' \in S$:*

$$\boldsymbol{P}^{emb(C)}(s,s') = \begin{cases} \dfrac{\boldsymbol{R}(s,s')}{E(s)} & \textit{if } E(s) \neq 0 \\ 1 & \textit{if } E(s) = 0 \textit{ and } s = s' \\ 0 & \textit{otherwise} \end{cases} \tag{2.5}$$

The above definition enables the CTMC's behaviour to be alternatively considered as staying in a state s for a delay exponentially distributed with rate $E(s)$, followed by a transition to state s' with probability given by $\mathbf{P}^{emb(C)}(s,s')$. To further allow for the analysis of the CTMC, the *infinitesimal generator matrix* must be extracted.

Definition 2.7. *The infinitesimal generator matrix for the CTMC $C = (S, s_0, \boldsymbol{R}, AP, L)$ is the matrix $\boldsymbol{Q} : S \times S \to \mathbb{R}$ defined as:*

$$\boldsymbol{Q}(s,s') = \begin{cases} \boldsymbol{R}(s,s') & \textit{if } s \neq s' \\ -\sum_{s'' \neq s} \boldsymbol{R}(s,s'') & \textit{otherwise} \end{cases} \tag{2.6}$$

In Section 2.2.1.1 we provided the definition of *paths* in a DTMC. Here we extend this definition to accommodate the needs of the continuous-time domain. Consequently,

a path π in a CTMC is a non-empty sequence of states $\pi = s_0, t_0, s_1, t_1, s_2, t_2, \ldots$ where $\mathbf{R}(s_i, s_{i+1}) > 0$ and $t_i \in \mathbb{R}_{>0}$ represents the time spent in state s_i, for all $i \geq 0$. State s_i is the i-th state of path π and is denoted by $\pi(i)$, t_i is the time spent in state s_i and is denoted by $time(\pi, i)$, and the occupied state at time t is denoted by $\pi@t$ and is given by $\pi(\omega)$, where ω is the smallest index for which $\sum_{i=0}^{\omega} t_i \geq t$. The set of all paths of C, starting in a given state s, is denoted $Path^C(s)$.

Analysing the behaviour of a CTMC entails identifying the probability of traversing specific paths. Based on the definition from [10], $C(s_0, I_0, \ldots, I_{n-1}, s_n)$ denotes a *cylinder set* consisting of all $Path^C(s)$ with non-empty intervals $I_0, \ldots, I_{n-1} \in \mathbb{R}_{>0}$, such that $\pi(i) = s_i$ for all $i \leq n$ and $time(\pi, i) \in I_i$ for all $i < n$. The probability measure (Pr_s) of the cylinder C is then inductively defined by $Pr_s(C(s_0)) = 1$ and

$$\begin{aligned} &Pr_s(C(s_0, \ldots, I_n, s_{n+1})) = \\ &\qquad Pr_s(C(s_0, \ldots, s_n)) \cdot \mathbf{P}^{emb(C)}(s_n, s_{n+1}) \cdot (e^{-E(s_n)\cdot\alpha} - e^{-E(s_n)\cdot\beta}) \end{aligned} \tag{2.7}$$

where $\alpha = inf\, I_n$ is the greatest lower bound, and $\beta = sup\, I_n$ is the least upper bound of the interval I_n.

Extending CTMCs with rewards

Similarly to DTMCs, CTMCs can be annotated with cost/reward structures that assign non-negative real-valued quantities to states and transitions. However, in contrast to DTMCs, the state costs/rewards are calculated based on the time spent in that state of the model. For instance, a reward of $t \cdot \rho(s)$ is obtained for $t \in \mathbb{R}_{\geq 0}$ time units that the model remained in state s.

Definition 2.8. *A cost/reward structure over a CTMC $C = (S, s_0, \boldsymbol{R}, AP, L)$ is a pair of real-valued functions $(\underline{\rho}, \iota)$ where:*

- $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ *is a state reward function that defines the rate $\rho(s)$ at which the cost/reward is obtained in a state $s \in S$ of C.*
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ *is a transition reward function that defines the value (cost/reward) $\iota(s, s')$ obtained each time a transition occurs from state $s \in S$ to state $s' \in S$.*

Example 2.2. *Figure 2.4 depicts a four-state CTMC modelling a queue of jobs, where state number i in state s_i indicates that there are i jobs in the queue. The graphical notation remains identical to that of DTMCs, with the only exception that transitions are now labelled with rates instead of probabilities. In this example, the queue is initially*

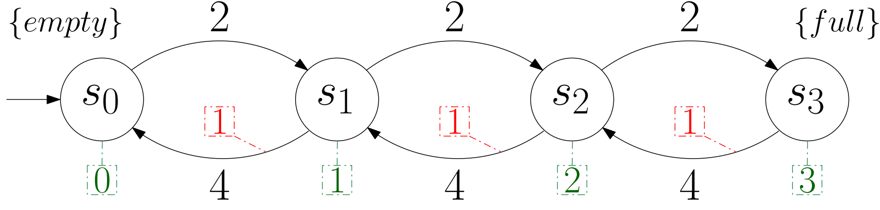


Fig. 2.4: A CTMC with four states modelling a queue of jobs, adapted from [134]

empty, and the maximum number of jobs it can hold is 3. Additionally, we assume a time unit of seconds and label each edge with the transition rate between states. Jobs arrive with a rate of 2, and are removed from the queue with a rate of 4. State s_0 is the initial state of the model, which corresponds to the queue being empty, and is labelled accordingly. The queue reaches its full capacity in state s_3. An incoming job request, considering that the queue is not full, is processed with a rate of 2 (Eqn. 2.4), i.e., the mean time spent in that state is $\frac{1}{2}$ seconds.

The CTMC is augmented with two different reward structures, whose values are shown in green and red dashed rectangular boxes, respectively. The first reward, indicated by green colour, associates with each state the number of requests that are awaiting service. This structure can be used when it is of interest to determine the queue size at any point in time or over extended time periods. The second reward, indicated by red colour, assigns 1 to the transitions corresponding to a request being served. Such a structure can be used for counting the number of served requests within a given time frame or for extended time periods.

The elements of the CTMC model are:

The set of states $S = \{s_0, s_1, s_2, s_3\}$

The initial state s_0

The set of atomic propositions $AP = \{empty, full\}$

The labelling function $L : L(s_0) = \{empty\}, L(s_3) = \{full\}$

The corresponding system matrices are:

$$\boldsymbol{R} = \begin{pmatrix} 0 & 2 & 0 & 0 \\ 4 & 0 & 2 & 0 \\ 0 & 4 & 0 & 2 \\ 0 & 0 & 4 & 0 \end{pmatrix} \quad \boldsymbol{P}^{emb(C)} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{2}{3} & 0 & \frac{1}{3} & 0 \\ 0 & \frac{2}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \boldsymbol{Q} = \begin{pmatrix} -2 & 2 & 0 & 0 \\ 4 & -6 & 2 & 0 \\ 0 & 4 & -6 & 2 \\ 0 & 0 & 4 & -4 \end{pmatrix}$$

and the reward structures are defined as:

$$\iota = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad \underline{\rho}^{queue_size} = (0, 1, 2, 3)$$

Finally, suppose that we want to identify the probability of leaving the initial state s_0 *and passing to state* s_1 *within the first 2 time units. This translates to the following sequence of states and intervals* 0, $[0,2]$ 1 *(i.e,* $s_0 = 0$, $I_0 = [0,2]$, *and* $s_1 = 1$*).*

Using the probability measure for the initial state s_0 *of the CTMC, for the cylinder set* $C(0,[0,2],1)$*, we have:*

$$\begin{aligned} Pr_{s_0}(C(0,[0,2],1)) =& 1 \cdot \boldsymbol{P}^{emb(C)}(0,1) \cdot (e^{-E(0)\cdot 0} - e^{-E(0)\cdot 2}) \\ =& 1 \cdot 1 \cdot (e^{0} - e^{-4}) = 1 - e^{-4} \simeq 0.981684 \end{aligned}$$

2.2.2 Probabilistic Temporal Logics

The next step following the synthesis of Markov chain model variants (e.g., DTMCs or CTMCs), representing the behaviour of a software system, is the quantification of the respective system's properties to enable reasoning whether requirements have been met. These properties represent nonfunctional aspects of a system, such as performance, reliability, response time, and cost, and they are formally expressed using probabilistic variants of temporal logic. The use of probabilistic temporal logics enables reasoning about the intended behaviour of a system as time progresses, and since transitions within Markov chain models are probabilistic, the interest falls in determining the likelihood of an event's occurrence, rather than on the event itself (i.e., if the event is true or not).

This section defines the syntax of Probabilistic Computation Tree Logic (PCTL) [8, 85] and Continuous Stochastic Logic (CSL) [7, 10], the probabilistic variants of temporal logic used to formalise the properties of DTMCs and CTMCs, respectively.

2.2.2.1 Probabilistic Computation Tree Logic

Probabilistic Computation Tree Logic (PCTL) [8, 85] is a probabilistic extension of the temporal logic CTL [42] that allows for quantification of probabilistic properties of a DTMC, i.e, extends CTL with a probabilistic operator P. Any property defined in PCTL references states and possible execution paths within a model D starting from an initial

state s_0. In this thesis, we use a cost-reward augmented PCTL variant with the syntax adopted from [104].

Definition 2.9. *The syntax of Probabilistic Computation Tree Logic (PCTL) is defined as follows:*

$$\Phi ::= true \mid a \mid \neg \Phi \mid \Phi \wedge \Phi \mid P_{\bowtie p}[\phi]$$

$$\phi ::= X\,\Phi \mid \Phi \cup^{\leq k} \Phi$$

and the cost/reward augmented PCTL state formulae are defined as:

$$R_{\bowtie r}[C^{\leq k}] \mid R_{\bowtie r}[I^{=k}] \mid R_{\bowtie r}[F\ \Phi]$$

where $a \in AP$ is an atomic proposition with AP being a set of atomic propositions, $\bowtie \in \{<, \leq, \geq, >\}$ is a relational operator, $k \in \mathbb{N} \cup \{\infty\}$, $p \in [0,1]$ is a probability bound (or threshold), and $r \in R_{\geq 0}$ is a reward bound.

In the definition above, Φ and ϕ denote state and path formulae, respectively. State formulae Φ are always used to specify properties of a DTMC, and path formulae ϕ can only occur within the scope of the probabilistic operator $P_{\bowtie p}[\cdot]$. For instance, a state s of a DTMC D satisfies $P_{\bowtie p}[\phi]$ if the probability of taking a path from s satisfying ϕ meets the bounds defined by $\bowtie p$. For a path π, the "next" formula $X\Phi$ holds if Φ is satisfied in the next state. The "bounded until" formula $\Phi_1 \cup^{\leq k} \Phi_2$ holds if Φ_2 is satisfied within k time-steps and Φ_1 is true up until that point. If $k = \infty$, the formula becomes "unbounded until", and $\Phi_1 \cup^{\leq k} \Phi_2$ is abbreviated by $\Phi_1 \cup \Phi_2$. Finally, formula $P_{=?}[\phi]$ can be used to obtain the probability of a path formula ϕ.

Given a state s in a DTMC D, the interpretation of cost/reward operator R is:

- $R_{\bowtie r}[C^{\leq k}]$ holds if the amount of the expected accumulated reward along a path π up to k time-steps meets the bound defined by $\bowtie r$;
- $R_{\bowtie r}[I^{=k}]$ is true if the expected state reward at k time-steps satisfies the bound defined by $\bowtie r$;
- $R_{\bowtie r}[F\ \Phi]$ holds if from state s the expected cumulative reward before reaching a state that satisfies Φ meets the bound $\bowtie r$.

The R operator, which works in a similar fashion to the P operator, can be used to quantify over states and transitions of a DTMC D, and to calculate the expected value of a cost/reward by using the formula $R_{=?}[\cdot]$.

The semantics of PCTL are defined formally over a DTMC D as follows:

Definition 2.10. *For any state $s \in S$ and path $\pi \in Path^D$ of a labelled DTMC $D = (S, s_0, \boldsymbol{P}, AP, L)$, the satisfaction relation $\models$ is inductively defined by:*

$$
\begin{aligned}
&s \models true && \forall\ s \in S \\
&s \models a && \Leftrightarrow a \in L(s) \\
&s \models \neg\Phi && \Leftrightarrow s \not\models \Phi \\
&s \models \Phi_1 \wedge \Phi_2 && \Leftrightarrow s \models \Phi_1 \wedge s \models \Phi_2 \\
&s \models P_{\bowtie p}[\phi] && \Leftrightarrow Prob^D(s, \phi) \bowtie p
\end{aligned}
$$

where

$$Prob^D(s, \phi) = Pr_s(\pi \in Paths^D(s) \mid \pi \models \phi)$$

is the probability that a path starting from s satisfies ϕ, and for any path $\pi \in Path^D(s)$:

$$
\begin{aligned}
&\pi \models X\Phi && \Leftrightarrow \pi(1) \models \Phi \\
&\pi \models \Phi_1 \cup^{\leq k} \Phi_2 && \Leftrightarrow \exists\, i \in \mathbb{N}.(i \leq k \wedge \pi(i) \models \Phi_2 \wedge \forall j < i.(\pi(j) \models \Phi_1))
\end{aligned}
$$

Finally, for the cost/reward structures:

$$
\begin{aligned}
&s \models R_{\bowtie r}[C^{\leq k}] && \Leftrightarrow Exp^D(s, \Psi_{C^{\leq k}}) \bowtie r \\
&s \models R_{\bowtie r}[I^{=k}] && \Leftrightarrow Exp^D(s, \Psi_{I^{=k}}) \bowtie r \\
&s \models R_{\bowtie r}[F\,\Phi] && \Leftrightarrow Exp^D(s, \Psi_{F\Phi}) \bowtie r
\end{aligned}
$$

where $Exp^D(s, \Psi)$ refers to the expectation of the random variable $\Psi : Path^D(s) \to \mathbb{R}_{\geq 0}$ with respect to the probability measure $\boldsymbol{Pr}_s$, and for any path $\pi = s_0 s_1 s_2 \cdots \in Path^D(s)$:

$$
\Psi_{C^{\leq k}}(\pi) := \begin{cases} 0 & \text{if } k = 0 \\ \sum_{i=0}^{k-1} \underline{\rho}(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases}
$$

$$
\Psi_{I^{=k}}(\pi) := \underline{\rho}(s_k)
$$

$$
\Psi_{F\Phi}(\pi) := \begin{cases} 0 & \text{if } s_0 \models \Phi \\ \infty & \text{if } \forall i \in \mathbb{N}.\ s_i \not\models \Phi \\ \sum_{i=0}^{min\{j \mid s_j \models \Phi\}-1} \underline{\rho}(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases}
$$

Example 2.3. *Consider again the DTMC model of Figure 2.3, representing a fictitious daily weather pattern. Table 2.1 shows a set of requirements for the weather model that*

Table 2.1: System requirements for the weather prediction DTMC model

Req.	Description	PCTL	Result
R1	The probability of reaching the *sunny* state without visiting the *cloudy* state is at most 35%	$P_{\leq 0.35}[\neg\text{"}cloudy\text{"}\ U\ \text{"}sunny\text{"}]$	25% ✔
R2	The probability of reaching the *cloudy* state within 20 time steps (NB: corresponding to 20 days) is greater than 68%	$P_{>0.68}[true\ U^{\leq 20}\ \text{"}cloudy\text{"}]$	97% ✔
R3	The expected time when the *sunny* state is eventually reached is less than 17 time steps (i.e., days)	$R^{\text{"}time\text{"}}_{<17}[F\ \text{"}sunny\text{"}]$	17.3 ✘

are used to quantify several requirements of interest. The requirements are expressed in both natural language and in PCTL.

2.2.2.2 Continuous Stochastic Logic

Continuous Stochastic Logic (CSL) [10, 7] is a variant of probabilistic temporal logic that is used to specify properties of CTMCs. CSL allows reasoning about path-based properties, similarly to PCTL with the use of the probabilistic operator P, and about steady-state, with the use of the steady-state operator S, and transient behaviour. The syntax of the cost-reward augmented CSL variant adopted in this thesis [105] is defined below.

Definition 2.11. *The syntax of a Continuous Stochastic Logic (CSL) state formula* Φ *and that of a path CSL formula* ϕ *are defined as follows:*

$$\Phi ::= true \mid a \mid \neg\,\Phi \mid \Phi \wedge \Phi \mid P_{\bowtie p}[\phi] \mid S_{\bowtie p}[\Phi]$$

$$\phi ::= X\,\Phi \mid \Phi \cup^{I} \Phi$$

and the cost/reward augmented CSL state formulae are defined as:

$$R_{\bowtie r}[C^{\leq t}] \mid R_{\bowtie r}[I^{=t}] \mid R_{\bowtie r}[F\ \Phi] \mid R_{\bowtie r}[S]$$

where $a \in AP$ *is an atomic proposition with* AP *being a set of atomic propositions,* $\bowtie\, \in \{<, \leq, \geq, >\}$ *is a relational operator,* $I \subseteq \mathbb{R}_{\geq 0}$ *and* $t \in \mathbb{R}_{\geq 0}$ *are a time interval and a time instant, respectively,* $p \in [0,1]$ *is a probability bound (or threshold), and* $r \in R_{\geq 0}$ *is a reward bound.*

The interpretation of CSL formulae is similar to that of PCTL formulae, with the exception of the interval $I \in \mathbb{R}_{\geq 0}$ parameter of the "until" operator U. For a path π, the "next" formula $X\Phi$ holds if Φ is satisfied in the next state. The "time-bounded

until" path formula $\Phi_1 \cup^I \Phi_2$ holds if Φ_2 becomes true at some time instant in the interval I, and Φ_1 holds at all previous time instants. If $I = [0, \infty)$, the formula becomes "unbounded until". The steady-state S operator describes the behaviour of the CTMC in the long-run, and thus, formula $S_{\bowtie p}[\Phi]$ asserts that the steady-state probability of being in a state s satisfying Φ meets the bound $\bowtie p$. Finally, the formula $P_{=?}[\phi]$ quantifies the probability of a path formula ϕ.

Given a state s in a CTMC C, the interpretation of cost/reward operator R is:

- $R_{\bowtie r}[C^{\leq t}]$ holds if the amount of the accumulated reward along a path π up to time t meets the bound defined by $\bowtie r$;
- $R_{\bowtie r}[I^{=t}]$ is true if the expected state reward at time instant t satisfies the bound defined by $\bowtie r$;
- $R_{\bowtie r}[F\ \Phi]$ holds if from state s the expected cumulative reward before reaching a state that satisfies Φ meets the bound $\bowtie r$;
- $R_{\bowtie r}[S]$ holds if the long-run average expected reward satisfies $\bowtie r$.

As in PCTL, the formula $R_{=?}[\cdot]$ can be used with CSL to obtain the expected value of a cost/reward.

The semantics of CSL are defined formally over a CTMC C as follows.

Definition 2.12. *For any state $s \in S$ and path $\pi \in Path^C$ of a labelled CTMC $C = (S, s_0, \boldsymbol{R}, AP, L)$, the satisfaction relation $\models$ is inductively defined by:*

$$\begin{aligned}
s &\models true && \forall\ s \in S \\
s &\models a && \Leftrightarrow a \in L(s) \\
s &\models \neg\Phi && \Leftrightarrow s \not\models \Phi \\
s &\models \Phi_1 \wedge \Phi_2 && \Leftrightarrow s \models \Phi_1 \wedge s \models \Phi_2 \\
s &\models P_{\bowtie p}[\phi] && \Leftrightarrow Prob^C(s, \phi) \bowtie p \\
s &\models S_{\bowtie p}[\Phi] && \Leftrightarrow \sum_{s' \models \Phi} Ps_s^C(s') \bowtie p
\end{aligned}$$

where

$$Prob^C(s, \phi) = Pr_s(\pi \in Paths^C(s) \mid \pi \models \phi)$$

is the probability that a path starting from s satisfies ϕ, and for any path $\pi \in Path^C(s)$:

$$\begin{aligned}
\pi &\models X\Phi && \Leftrightarrow \pi(1) \models \Phi \\
\pi &\models \Phi_1 \cup^{\leq I} \Phi_2 && \Leftrightarrow \exists\, t \in I.(\pi@t \models \Phi_2 \wedge \forall t' \in [0, t).(\pi@t' \models \Phi_1))
\end{aligned}$$

Table 2.2: System requirements for the queue of jobs CTMC model

Req.	Description	CSL	Result
R1	The probability of reaching the *full* state of the queue within 8 seconds must be lower than 70%	$P_{<0.7}[true\ U^{[0,8]}\ \text{"}full\text{"}]$	77% ✘
R2	The expected queue size in the first 5 seconds is less than 2	$R^{\text{"}queue_size\text{"}}_{<2}[I^{=5}]$	0.73 ✔
R3	The expected number of requests served before the queue becomes full is greater or equal to 2	$R^{\text{"}served_reqs\text{"}}_{\geq 2}[F\ \text{"}full\text{"}]$	7.99 ✔

and $Ps^C_s(s')$ *is the steady-state probability, i.e., the probability of finally reaching state* s' *from a state* s *(in the long-run).*

Finally, for the cost/reward structures:

$$
\begin{aligned}
s &\models R_{\bowtie r}[C^{\leq t}] \Leftrightarrow Exp^C(s, \Psi_{C^{\leq t}}) \bowtie r \\
s &\models R_{\bowtie r}[I^{=t}] \Leftrightarrow Exp^C(s, \Psi_{I^{=t}}) \bowtie r \\
s &\models R_{\bowtie r}[F\ \Phi] \Leftrightarrow Exp^C(s, \Psi_{F\Phi}) \bowtie r \\
s &\models R_{\bowtie r}[S] \quad \Leftrightarrow lim_{t\to\infty} \frac{Exp^C(s, \Psi_{C^{\leq t}})}{t} \bowtie r
\end{aligned}
$$

where $Exp^C(s, \Psi)$ *refers to the expectation of the random variable* $\Psi : Path^C(s) \to \mathbb{R}_{\geq 0}$ *with respect to the probability measure* Pr_s*, and for any path* $\pi = s_0 t_0 s_1 t_1 s_2 t_2 \cdots \in Path^C(s)$*:*

$$
\begin{aligned}
\Psi_{C^{\leq t}}(\pi) &:= \sum_{i=0}^{j_t-1} \left(t_i \cdot \underline{\rho}(s_i) + \iota(s_i, s_{i+1})\right) + \left(t - \sum_{i=0}^{j_t-1} t_i\right) \cdot \underline{\rho}(s_{j_t}) \\
\Psi_{I^{=t}}(\pi) &:= \underline{\rho}(\pi @ t) \\
\Psi_{F\Phi}(\pi) &:= \begin{cases} 0 & if\ \pi(0) \models \Phi \\ \infty & if\ \forall i \in \mathbb{N}.\ s_i \not\models \Phi \\ \sum_{i=0}^{min\{j \mid s_j \models \Phi\}-1} t_i \cdot \underline{\rho}(s_i) + \iota(s_i, s_{i+1}) & otherwise \end{cases}
\end{aligned}
$$

Example 2.4. *Consider again the CTMC model of Figure 2.4, representing a queue of jobs. Table 2.2 shows a set of system requirements for this queue, expressed in both natural language and CSL.*

2.2.3 Probabilistic Model Checking Tools and Applications

The introduction of the first probabilistic model checking algorithms was made in the 1980s; however, the first "*industrial strength probabilistic model checkers*" appeared later in the early 2000s [106]. The development of suitable probabilistic model checking tools enabled the practical application of these techniques. Today, there is a wide range of such tools, some of which are summarised below.

PRISM [107, 109] is a tool aiming to model and analyse systems that exhibit random or probabilistic behaviour. It was initially developed at the University of Birmingham and now shares development and extensions with the University of Oxford. PRISM can build and analyse several types of probabilistic models such as discrete-time Markov chains, continuous-time Markov chains, and Markov decision processes (MDPs). Models are described using the PRISM state-based language, and a wide range of quantitative properties expressed in PCTL or CSL can be analysed.

Storm [60, 86] is another probabilistic model checker that has been developed at Aachen University and went open-source in 2017. Some of its main characteristics are its modular set-up, which enables the exchange of different solvers, and its focus on performance by providing a good space-time tradeoff. Furthermore, it offers support for several major input languages, including PRISM's state-based language and the specification of properties in PCTL and CSL.

Another notable mention is the Markov Reward Model Checker (MRMC) [96], which was developed by the Formal Methods & Tools (FMT) group and the Software Modeling and Verification (MOVES) group at the University of Twente and Aachen University, respectively. It is a command-line tool written in the C programming language, based on a sparse matrix representation. Similarly to the previous tools, it supports PCTL and CSL model checking, as well as their reward extensions. Prominent features of MRMC include quantification of time- and reward-bounded reachability probabilities, bisimulation minimisation, and precise steady-state detection.

PRISM and MRMC provide support for all three major operating systems, while Storm is only supported in Mac OS X and Linux. As far as models are concerned, all tools support DTMCs and CTMCs, but exhibit differences in the rest of the supported models. PRISM supports probabilistic automata (PAs) and probabilistic timed automata (PTAs), Storm supports Markov automata (MAs), and MRMC supports continuous-time Markov decision processes (CTMDPs), unlike PRISM and Storm, which support Markov Decision processes (MDPs).

Table 2.3: Summary of probabilistic model checking tools

	PRISM	Storm	MRMC
Year of release	2000	2017	2005
Platform support	Windows, Linux, Mac OS X	Linux, Mac OS X	Windows, Linux, Mac OS X
Model support	DTMC, CTMC, MDP, PA, PTA	DTMC, CTMC, MDP, MA	eDTMC, CTMC, CTMDP
GUI	Yes	No	No
Performance	Good memory usage and speed even for larger models	Most efficient option out of the three listed	Good option for small models
Other features	Models can easily be created and exported to other tools	Supports several types of input, provides a Python API	Simple input format, more appropriate as back-end verification engine

One of the major aspects that determines tool selection between alternatives among both research community and industry is performance. Jansen et al., in their 2008 paper [92], performed a comparison between 5 probabilistic model checking tools and based on these findings, we can make the following observations. PRISM has good memory usage and speed overall, but the area where it excels the most is in dealing with larger models. On the other hand, MRMC is better at analysing smaller models, but when dealing with larger, more complex models, it becomes significantly slower. Storm was not available at the time of this comparison, but results obtained in the analysis performed by Dehnert et al. in their 2017 paper [60], showed that Storm outperforms its rivals in most cases.

In this thesis, both PRISM and Storm were chosen as the tool of choice depending on the scenarios we faced. We used PRISM for constructing models, specifying and verifying properties when modelling and verification were done manually, due to the ease of using its graphical user interface (GUI), and Storm when we were dealing with larger models, and modelling and verification were automated. Table 2.3 summarises the key characteristics of each of the mentioned probabilistic model checkers.

Probabilistic model checking has seen practical application in various domains and has been successfully used to analyse real-world problems. Examples of such domains include, but are not limited to, reliability engineering for safety-critical systems, dependability modelling and analysis of spacecraft systems, performance modelling techniques, and systems biology [95]. However, challenges still remain [108], mainly related to the accuracy of model representations of real systems [128, 129]. Predicting

real-world behaviours is not a trivial task, and existing tools can only effectively support the design and verification of these systems when accurate model representations exist.

2.3 Performance Antipatterns

Software patterns [46, 72] are considered "*abstractions of positive experience*" [141], and essentially are documented solutions to problems that not only occur in various contexts, but also have the same underlying causes. Antipatterns [24, 25], which are conceptually similar to patterns in the sense that they document solutions to recurring design problems, capture common mistakes made during software development along with ways to prevent them. The definition of an antipattern includes the problem and its solution (i.e., actions to solve the problem). Some of these antipatterns have their roots in poor software development practices, some in hardware-related issues, and some are just common mistakes made during the design or development phases. Antipatterns are meant to be refactored (based on their documented solution) to overcome the addressed problem. Refactoring is an effective approach for improving the software's quality, while maintaining its correctness [24].

In this thesis, we are interested in using antipatterns from a performance perspective. Research on defining performance antipatterns is mainly reflected by the work of Smith and Williams across the years [144, 146, 147]. An overview of the current performance antipatterns, adopted from [158], can be seen in Table 2.4. From this list, we focus on the definition of the three highlighted antipatterns based on [144, 146]. These antipatterns are used later on in Chapter 4 to identify performance problems in our case studies and guide refactoring.

THE "GOD" CLASS (OR BLOB[1])

Problem *- A single class of the system monopolises the processing, leading to relegation of the other classes to minor roles. Another BLOB variant, instead of monopolising the processing contains all system data. Both variants cause an increased exchange of messages to perform any function that results in degradation of the system's performance.*

In the case of the first BLOB variant, the system contains a complex *controller* class that performs most of the processing while the rest of the classes primarily encapsulate data. These classes usually contain only accessor functions (i.e., *get()* and *set()* methods)

[1]The name BLOB has been given to the "GOD" class antipattern after the analogy to the monster from the 1958 film *The Blob*, which absorbed anything it touched and grew larger and larger over time. The BLOB antipattern has been infamous for such behaviour with respect to object-oriented architectures.

Table 2.4: Overview of performance antipatterns, adopted from [158]

Antipattern	Problem	Solution
Circuitous Treasure Hunt	Search in a database requires a large amount of processing	Refactor the design to enable alternative access paths
Concurrent Processing Systems	Inefficient use of available processors	Balance the system's load or enable multi-threading
Empty Semi Trucks	Excessive number of requests required to perform a task	Use the Batching and Coupling performance patterns, and the Session Facade and Aggregate Entity design patterns [147]
Excessive Dynamic Allocation	Unnecessary creation and destruction of same class objects within an application	1) Recycle objects when needed 2) Use the Flyweight pattern to prevent creating new objects [144]
Extensive Processing	A long running process monopolizes a processor	Restructure, re-order and/or remove the processing steps
More is Less	"Thrashing" caused by too many processes being relative to the available resources	Quantify the thrashing thresholds and determine if the architecture can meet its performance goals while staying below the thresholds
One-lane Bridge	One or few processes can execute concurrently while the other wait their turn	Use the shared resources principle [143] to minimize conflicts
"Pipe & Filter" Architectures	Slowest filter negatively affects throughput	Reduce overhead by dividing long and combining short filters
The "God" Class	Excessive messaging leading to performance degradation	Uniform distribution of intelligence
The Ramp	Processing time increases as the system is used	Select self-adapting algorithms based on the size of data
Tower of Babel	Excessive translation of information into an exchange format, such as XML	Use the Fast Path and Coupling performance patterns [147]
Traffic Jam	A large backlog of jobs waiting for service produces wide variability in response time	Eliminate the original cause of the backlog or provide sufficient power to handle the worst-case load

and do not take part in any major processing operations. On the other hand, the BLOB *controller* class obtains information from the other classes using their *get()* methods, performs computations, and then updates the data using their *set()* methods.

In the case of the second BLOB variant, a single class contains most of the data of a system, and all other functions are assigned to the rest of the classes. When a data-driven operation needs to be performed from any of the other classes, the BLOB class supplies them with the required data, i.e., they contact the BLOB class via their *get()* and *set()* methods to obtain and update data, respectively.

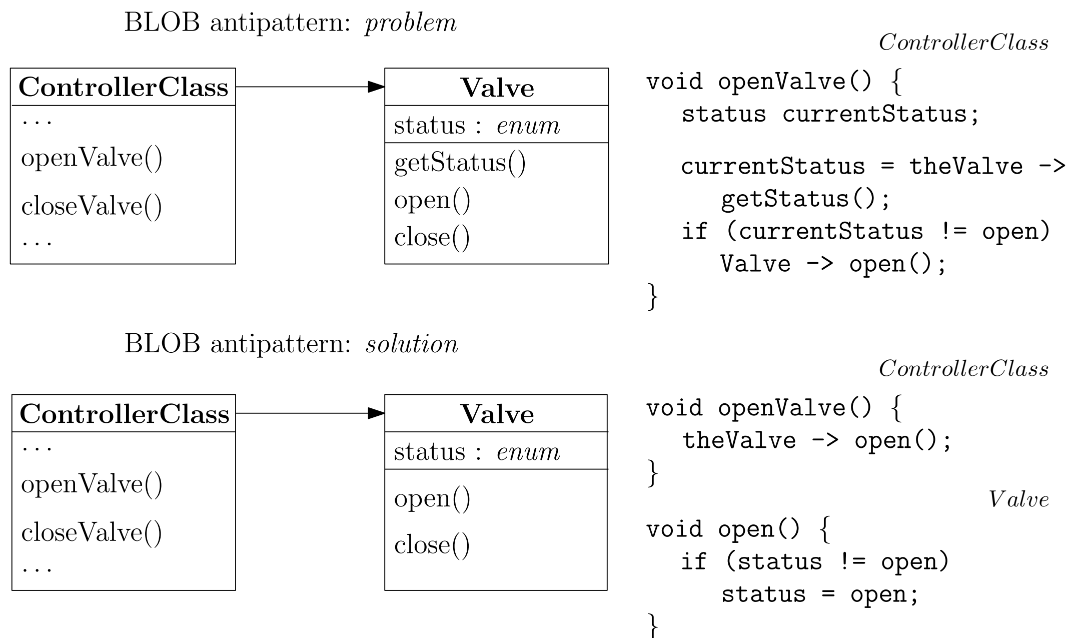


Fig. 2.5: BLOB in industrial process control application, adapted from [144]

The "GOD" class is the product of a poorly distributed system intelligence. In object-oriented systems, this is a result of distributing related data and system behaviour in different classes or components. In both BLOB cases, performing a function requires a larger amount of exchanged messages in comparison to a design that assigns related data and behaviour to the same class. The excessive message traffic between the BLOB entity and the rest of the classes impacts the system's performance.

Solution - *Refactor the system design to uniformly distribute intelligence over the top-level classes, and keep the related data and behavior within the same class.*

The BLOB antipattern's solution focuses on refactoring the system design, with an emphasis on keeping related data and behavior together. The performance gain for the refactored solution is defined by $T_s = M_s \times O$, where T_s is the saved processing time, M_s is the number of saved messages and O is the mean overhead per message. The amount of a message's overhead depends on the call's type (e.g., a remote procedure call has more overhead than a local call).

Example 2.5. *The first class diagram from Figure 2.5 illustrates a possible design for an industrial process control application, adapted from [144]. In this application, it is required to control the status of a valve which is either open or closed. The* `ControllerClass` *behaves like a BLOB entity as it performs all of the work. The* `Valve` *class, on the other hand, has no intelligence and its main purpose is to report its status and respond to the open() and close() method invocations. The first code fragment*

in Figure 2.5 shows that the `ControllerClass` *is inordinately linked to the* `Valve` *class, and requires additional messages to perform any of its simple operations. E.g., to open the valve, the controller must first request the valve's status, check if the valve is open, and then proceed with opening the valve.*

To solve the problem of `ControllerClass` *behaving like a BLOB entity, refactoring is needed to reduce the coupling, and the amount of exchanged messages between the two classes. This is achieved by moving the status check operation inside the* `Valve` *class (as seen in the second class diagram and code fragments). The status check operation now takes place inside the appropriate class, i.e., the one that contains the required data to perform the check.*

The following two antipatterns are manifestations of the UNBALANCED PROCESSING antipattern [146]. This antipattern and its manifestations are linked to problems arising during concurrent processing. A common cause of such problems is when multiple threads wait for other processing to complete, which impacts scalability.

CONCURRENT PROCESSING SYSTEMS (CPS)

Problem - *The system's processes cannot effectively use the available processors.*

The unbalanced processing occurrence is due to the system's processes inability to use the available processors effectively. This is a result of either an inefficient distribution of tasks that leads to the unavailability of some processors or because the code is single-threaded. In the CPS manifestation, it is necessary to ensure that the system is able to use the available processors.

Solution - *Use system execution and performance models to assess the load balance of threads to processors, and identify alternatives that improve the performance of the system.*

If the problem results from an unbalanced distribution of threads to processors, the solution is to use system execution models to quantify the net effect of each assignment. Based on the outcome of the quantification, the engineer can then restructure the system's components to balance the load. If the problem results from single-threading work, the solution is to use performance models to identify alternatives that enable multi-threading or execute concurrently multiple copies of the same process if possible.

Example 2.6. *Consider the application of the CPS antipattern on the system depicted in Figure 2.6, adapted from [146]. The violation of performance requirements is caused by a routing algorithm that is based on static properties, resulting in an unbalanced distribution of tasks, i.e., a queue is heavily targeted with tasks sent from the routing*

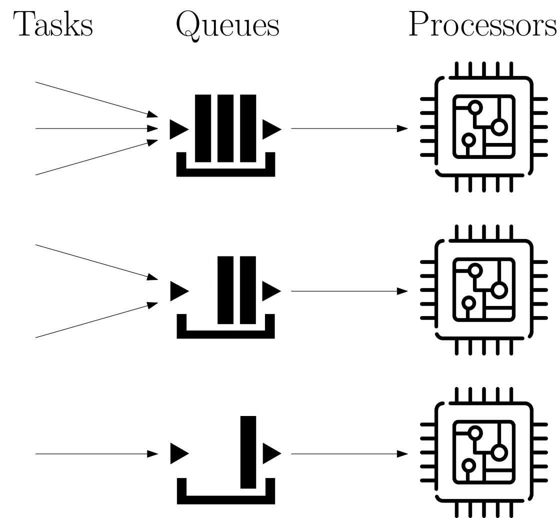


Fig. 2.6: Unbalanced processing caused by routing algorithm, adapted from [146]

algorithm compared to the overload of the other queues. The antipattern's solution is to use a dynamic algorithm instead that assigns tasks to queues based on the requirements of each task and the system's congestion.

"PIPE AND FILTER" ARCHITECTURES (P&F)

Problem - *The slowest filter (component) in a "Pipe and Filter" architecture determines the overall throughput of the system, violating the respective system requirement for throughput.*

The overall system throughput is determined by the slowest filter and deemed unacceptable by the system's performance requirements. In more detail, a bottleneck is formed due to a filter in the system's pipeline being significantly slower than all the others. Only when the slowest filter terminates, the remaining filters can resume or initiate their workload.

Solution - *Use modeling techniques to determine the system's processing requirements and apply refactoring actions, targeting the filter(s) responsible for the violation of these requirements.*

The solution to the P&F antipattern is twofold. Dividing long processing filters into multiple, smaller ones that can execute in parallel to reduce processing time, and combining short processing filters to reduce context switching overhead and other delays for shared resources.

Example 2.7. *Consider the P&F architecture from Figure 2.7, adapted from [146]. The application of the antipattern can inform the system's engineers that unbalanced*

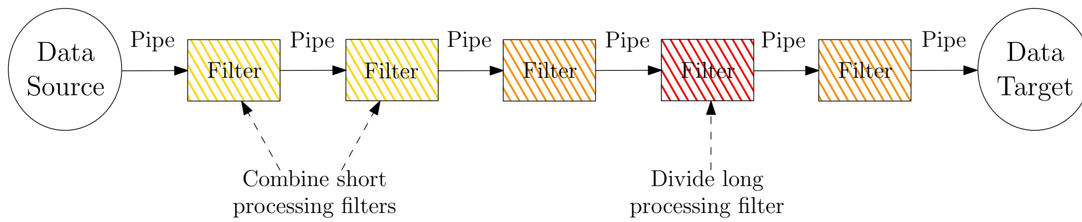


Fig. 2.7: Unbalanced processing in a P&F architecture, adapted from [146]

processing is caused due to filters of unmatched sizes, as shown in the execution graph of the figure. The colour for each filter indicates its relative processing time. Yellow colour reflects small processing time, orange colour reflects normal/expected, as per the system's requirements, processing time, and red colour reflects long processing time. The solution obtained from this antipattern analysis is to combine the two filters with short processing times to reduce overhead, and divide the filter with long processing time into multiple smaller instances that can execute in parallel, thus improving performance (as indicated by the notes at the bottom of Figure 2.7).

2.4 Summary

This chapter defined key elements of existing work focusing on system modelling and analysis techniques, which enable the formal verification of system models, and allow reasoning about nonfunctional system requirements (e.g, performance). Section 2.2 introduced the concept of probabilistic model checking, whose application can be found in all main chapters of this thesis, and focuses on the verification of nonfunctional properties of systems. Specifically, Section 2.2.1 and Section 2.2.2 defined the two types of probabilistic models (DTMCs and CTMCs) used in our work, and the probabilistic temporal logics (PCTL and CSL) that allow for the formal specification of properties of interest over these types of models, respectively. Additionally, we provided an overview of the most popular probabilistic model checking tools used by both the research community and industry in Section 2.2.3. Finally, in Section 2.3 we introduced the concept of performance antipatterns, along with definition of the types of antipatterns used in Chapters 4 and 5 of this thesis.

Chapter 3

Probabilistic Analysis of Code Performance

This chapter introduces a probabilistic program performance analysis (PROPER) method that circumvents a major drawback of current software performance analysis approaches at code-level, i.e, the need for executing the code for every platform and usage profile of interest, and after every change in the code (as described in Section 1.1). To this end, we automatically derive a discrete-time Markov chain (DTMC) model of the analysed code, exploiting usage profile information from program logs to calculate the model's transition probabilities. Performance concerns such as the execution time or energy use of individual statements or library function calls are encoded as DTMC reward structures, and the program performance properties of interest are formalised in probabilistic temporal logic and evaluated through the probabilistic model checking of this DTMC. PROPER supports the what-if analysis of program performance in several scenarios of practical importance:

- before deploying the code on a new platform;
- for an expected change in the usage profile of the software;
- to assess the performance impact of using a new implementation of a function called by the analysed code.

As discussed later in Section 3.5, PROPER is the first method that uses probabilistic model checking to automatically evaluate software performance properties at code level. An approach that uses probabilistic modelling for code-level analysis was proposed in [69, 70]. However, unlike our PROPER method, this approach addresses the analysis

of program reliability, uses bounded loop unfolding to handle loops, and therefore can only perform approximate analysis for programs that contain loops.

The main contributions of the work presented in this chapter are:

- the theoretical foundation underpinning the generation of the PROPER Markov-chain models;
- a prototype tool that implements our theoretical results, automating the PROPER synthesis of DTMC models for the performance analysis of Java methods;
- an extensive evaluation of the PROPER method and tool for code from an existing Java library, Android application, and optimisation algorithm implementation.

We organised the rest of the chapter as follows. Section 3.1 introduces a running example used to illustrate the application of our performance analysis method. Section 3.2 presents the PROPER theoretical foundation and our prototype tool, starting with a brief highlighting of the key steps of the approach. In the first step (Section 3.2.1), a reward-augmented DTMC model is automatically extracted from the Java code of interest. The second step (Section 3.2.2), involves the calculation of the transition probabilities associated with states in the generated DTMC that model the conditional statements and loops from the code. In the third and final step (Section 3.2.3), probabilistic model checking is applied to the DTMC model to analyse performance properties of interest. Finally, we present the evaluation of our program performance analysis method in Section 3.4, we discuss related research in Section 3.5, and we conclude with a brief summary in Section 3.6.

3.1 Motivating Example

To illustrate the steps and application of our PROPER method, we consider the `distance1` Java method from the Apache Commons Math library.[1] This method calculates the L1 distance between two points in multidimensional space, which is a distance metric widely used in applications such as machine learning. As shown in Figure 3.1, the method receives as input two integer arrays, and checks whether the arrays have equal length in line 3. An exception is thrown if the arrays have different lengths (line 4). Otherwise, the absolute distance between the points is calculated using the *Math.abs* function (line 11) and is returned in line 14.

[1] https://commons.apache.org/proper/commons-math/

```
1  public static int distance1(int[] p1, int[] p2)
2                  throws DimensionMismatchException {
3    if (checkEqualLength(p1, p2) == false) {
4      throw new DimensionMismatchException
5                    (p1.length, p2.length); // @cost=7
6    }
7    else {
8      int sum = 0;
9      int i = 0;
10     while (i < p1.length) {
11       sum += Math.abs(p1[i]-p2[i]); // @time=2.5
12       i++;
13     }
14     return sum;
15   }
16 }
```

Fig. 3.1: Java method `distance1` from the Apache Commons Math library

We suppose that the method `distance1` is used by an application for which a detailed log reflecting the method's usage profile (i.e., the typical combinations of argument lengths that `distance1` is invoked with) is available. Additionally, we suppose that the application's developers want to assess:

- the expected cost (i.e., the mean cost) for an invocation of the method, given that a cost of 7 is incurred each time when the method throws an exception in line 4;
- the method's expected execution time, if each execution of the statement from line 11 requires 2.5ns on average.

The annotations '@*cost*=*7*' and '@*time*=*2.5*' appended as comments to lines 5 and 11, respectively, are used to specify the two performance properties whose evaluation is of interest. We assume that the rest of the lines of code from Figure 3.1 take a negligible amount of time to be executed; thus, they are not annotated with a '*time*' property.

Instead of annotating each relevant statement of a block of code with a property value, it is possible to annotate the block of code with the sum of these values. However, this limits the information that the user can extract about specific lines in the code.

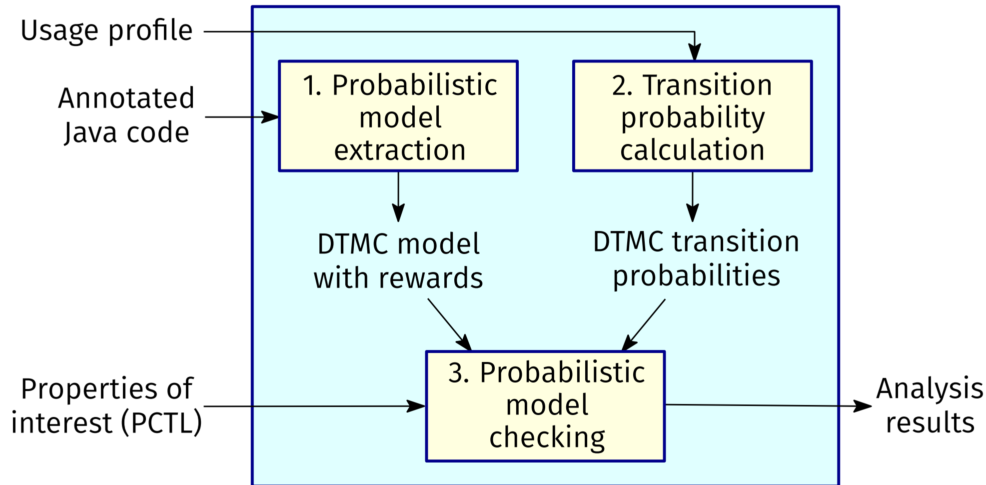


Fig. 3.2: PROPER program performance analysis

3.2 Approach

As shown in Figure 3.2, PROPER carries out the analysis of the performance properties of a program in three steps. In the first step, a reward-augmented DTMC model is automatically extracted from the analysed Java code. To that end, the code is first annotated with the performance properties of interest by appending a comment of the form

$$// \ @property{=}value \tag{3.1}$$

to the Java statements that these performance properties are associated with. In this PROPER annotation, *property* can be any one-word label (e.g., '*cost*' or '*time*', as shown in Figure 3.1), and *value* is a positive quantity such as 7 or 2.5. The same property label can be added to as many statements as required, e.g., to indicate that a non-negligible *cost* or execution `time` is associated with multiple statements.

The second PROPER step calculates the transition probabilities associated with the DTMC states that model the conditional statements and the loops from the code. This calculation is carried out based on the usage profile of the analysed code, taken or derived from program logs, where we assume that the code is appropriately instrumented to generate logs containing this information.

The first two steps can take place in parallel as seen in Figure 3.2. The transition probabilities can be calculated directly from the usage profile of the analysed code (see Section 3.2.2) and then be added to the resulting DTMC model.

In the third PROPER step, the performance properties of interest, specified in PCTL, are analysed by applying probabilistic model checking to the DTMC model obtained in

step 1. To enable this analysis, the transition probabilities of the DTMC are set to the probability values calculated in step 2.

The three steps of our method and further types of analyses enabled by the DTMC model are described in detail in the remainder of this section. The PROPER method is applicable to the performance analysis of single-threaded Java code. The current version of our PROPER prototype tool can handle the analysis of single Java methods that use variables declared locally or passed as arguments to the method, and whose invocations of other methods have no side effects (i.e., do not change the analysed method's variables). However, these constraints are only a limitation of the current implementation: the steps of our method do not impose any of these constraints.

The reason for choosing Java as the programming language of choice for the first PROPER step is due to the popularity and application of the language in various domains, such as desktop computing [65], mobile computing [38, 87] and numeric computing [118], among others. For instance, *WorldWind*[2] is a NASA desktop computing application developed in Java. There is even notable research work aiming at bringing to safety-critical industry the possibility of using Java [113, 117]. According to the index published by the TIOBE Programming Community[3], Java has consistently held the number-one slot between 2004 and 2020 [155] for the most used programming language. The selection of Java does not limit the application of the approach, which can be used with any other object-oriented language, such as C++ or Python. The translation of programming constructs into DTMCs for these languages would follow the same transformation principles. Additionally, PROPER could also be used with procedural languages, such as C.

PROPER supports the reuse of existing probabilistic models in scenarios where only the usage profile of the application has changed, and there are no major changes or introduction of additional code. Changes in the usage profile correspond to changes in the transition probability calculation (i.e., updating the transition probabilities in the model). If the value of a property derived from an annotation in the code needs to be updated, it can easily be done by modifying the model's rewards structures only, leaving the probabilistic structure of the Markov chain (states and transition probabilities) unchanged. Some cases of minor changes in the code can still be addressed without

[2]A geographic information system that provides graphical access to terabytes of imagery and elevation models for planets and other celestial objects (https://worldwind.arc.nasa.gov/).

[3]The TIOBE Programming Community index is an indicator of the popularity of programming languages (https://www.tiobe.com/tiobe-index/).

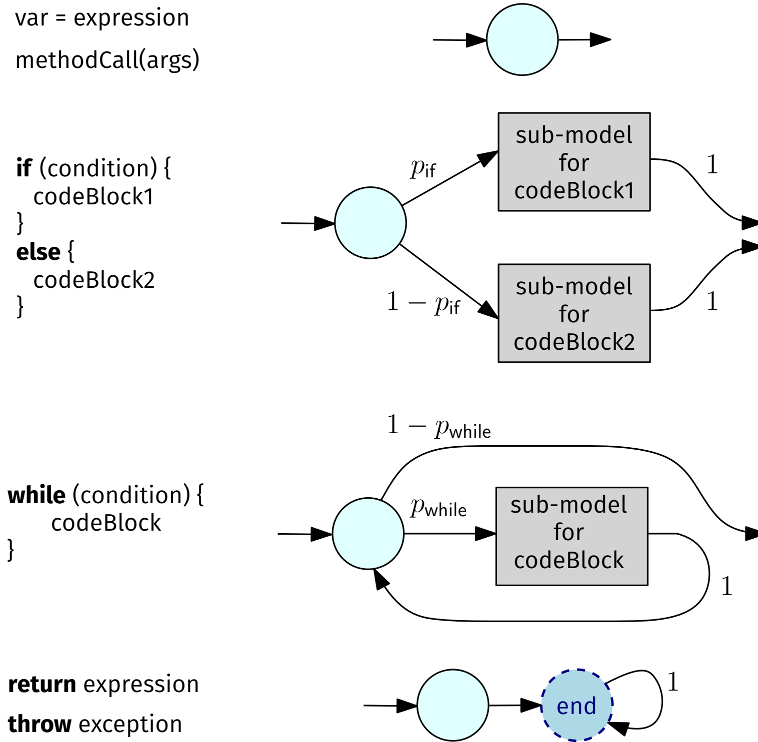


Fig. 3.3: PROPER code-to-model transformation rules

reapplying the model extraction step, and examples of such cases are presented in Section 3.4.

3.2.1 Probabilistic Model Extraction

The synthesis of the DTMC model is carried out by recursively applying the code-to-model transformation rules from Figure 3.3. We distinguish between four types of statements:

1. Assignment statements and method calls (with no side effects) are modelled using a single DTMC state. This state has one incoming transition (from the DTMC fragment modelling the previous statement in the code) and one outgoing transition (to the DTMC fragment modelling the next statement).

2. Conditional statements are modelled using a state with two outgoing transitions, one to the DTMC fragment modelling the statements from the 'if' branch, and one to the DTMC fragment modelling the 'else' branch. The latter DTMC fragment is empty if the else branch is missing. The derivation of the probability p_{if} from the program logs is described in the next section.

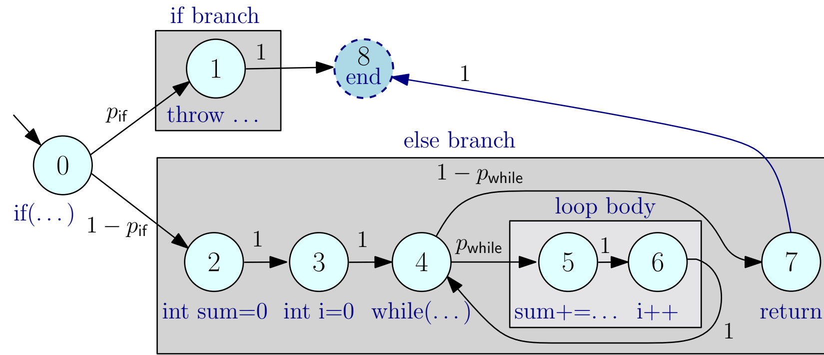


Fig. 3.4: DTMC model for the `distance1` Java method

3. Loops are modelled using a state with two outgoing transitions, one leading to the DTMC fragment modelling the statements from the loop body, and one leading to the fragment modelling the statement that comes after the loop. Additionally, the outgoing transition of the DTMC fragment modelling the statements from the loop body leads back to the initial state of the loop. The derivation of the probability p_{while} for the initial state of the loop is described in the next section. Note that we only focus on 'while' loops since other types of loops (e.g., 'for' loops) can easily be converted into 'while' loops. PROPER supports the quantitative analysis of the *expected* value of a random variable[4] associated with a property of interest, i.e., the first moment of the distribution for this property; this value is computed by modelling the distribution with a geometric distribution with the same mean.

4. Return statements and exceptions are modelled using a state whose only outgoing transitions leads to the "end" state of the DTMC. This state, shown in dashed line in Figure 3.3, has a self-loop transition of probability 1, does not correspond to any statement from the code, and is used as the sink state for all outgoing transitions corresponding to final statements from the code.

Example 3.1. *Figure 3.4 shows the DTMC obtained by applying these rules to the Java code from our running example. The statement modelled by each DTMC state is mentioned under the state, and the states are numbered* 0 *to* 8.

To allow the use of model checkers to analyse its synthesised DTMCs, PROPER uses the rules from Figure 3.3 to generate these DTMCs in the high-level modelling language of the PRISM model checker [109], which models a system as the parallel

[4]Note that the property is analysed as a random variable [37, 165, 169]. It represents the expected total amount of reward cumulated (over a number of steps, until a set of states is reached, or indefinitely) or the expected value of a reward structure at a particular instant. A formal definition can be found in Section 2.2.2.1.

composition of a set of modules. The state of a module is determined by a set of finite-range local variables, and its state transitions are specified by probabilistic guarded commands that modify these variables, and have the form:

$$[action]\; guard \rightarrow\; e_1 : update_1 \;+\; \ldots \;+\; e_n : update_n;$$

where *guard* is a boolean expression over all model variables. If the *guard* is true, the arithmetic expression $e_i, 1 \leq i \leq n$, gives the probability with which the $update_i$ change of the module variables occurs. When the optional label *action* is present, all modules comprising commands with the same *action* must perform one of these commands simultaneously.

The DTMC produced by PROPER comprises a single PRISM module, and is generated by the function BUILDMODEL from Algorithm 1. This function takes as input a Java method, parses its code into an abstract syntax tree *ast* in line 33, and obtains the PRISM module commands by invoking the function SYNTHESIS. These commands—prefixed with the appropriate model preamble assembled in lines 35 and 36, and followed by the model ending built in line 37—are then returned in line 38. PROPER enables the automated mapping between statements in the code and states in the resulting DTMC model.

SYNTHESIS starts with a *model* comprising an empty sequence of commands (line 3). The *model*'s guarded commands are then generated by the for loop in lines 4–29. The iterations of this loop handle one statement from the *ast* abstract syntax tree at a time, by using the switch from lines 5–25 to handle each statement according to its type. The four cases of the switch statement correspond to the four types of statements described earlier in this section. This part of the algorithm uses the counters *stateCtr* and *condCtr* (initialised in line 1) to keep track of the index for the states and transition probabilities being generated, respectively.

A single guarded command is generated if the processed statement *stmt* is an assignment or a method call (line 7). If *stmt* is a conditional, a new state with two outgoing transitions is created (line 9). The first transition, corresponding to the 'if' branch of the conditional, points to the next state with a probability $\mathrm{p}_{condCtr}$. The second transition, corresponding to the 'else' branch (if this branch exists) or to the statement after the conditional (otherwise), has probability $1-\mathrm{p}_{condCtr}$, points to a state identified (in line 12 if the else branch is missing, or in line 14 otherwise) after the *model* commands for the 'if' branch are obtained by invoking SYNTHESIS recursively in line 10. These commands are appended to the *model* in line 12 if the 'else' branch is

Algorithm 1: DTMC model synthesis (shaded strings indicate literals included in the model)

```
1  stateCtr=0, condCtr=0, rewards = ()
2  function SYNTHESIS (ast)
3      model = ‘’
4      for each stmt ∈ ast do
5          switch (stmt)
6              case assignment or methodCall :
7                  model += ‘[] s=’ + (stateCtr++) + ‘→ 1:(s′=’ + (stateCtr) + ‘);’
8              case conditional :
9                  model += ‘[] s=’ + (stateCtr++) + ‘→ p’ + condCtr + ‘:(s′=’ +
                     (stateCtr) + ‘)+(1-p’ + (condCtr++) + ‘):(s′=’
10                 if_branch_model = SYNTHESIS (stmt.thenStmts) ;
11                 if ¬stmt.hasElseBranch then
12                     model += (stateCtr) + ‘);’ + if_branch_model
13                 else
14                     model += (++stateCtr) + ‘);’ + if_branch_model + ‘[] s=’ +
                         (stateCtr − 1) + ‘→ 1:(s′=’
15                     else_branch_model = SYNTHESIS (stmt.elseStmts)
16                     model += (stateCtr) + ‘);’ + else_branch_model
17                 end
18             case loop :
19                 loopStartingState=stateCtr
20                 model += ‘[] s=’ + (stateCtr++) + ‘→ p’ + condCtr + ‘:(s′=’ +
                     (stateCtr) + ‘)+(1-p’ + (condCtr++) + ‘):(s′=’
21                 loop_body_model = SYNTHESIS (stmt.loopBody)
22                 model += (++stateCtr) + ‘);’ + loop_body_model + ‘[] s=’ +
                     (stateCtr − 1) + ‘→ 1:(s′=’ + loopStartingState + ‘);’
23             case return or exception :
24                 model += ‘[]s=’ + (stateCtr++) + ‘→1:(s′=end_state);’
25         end
26         while reward = stmt.getNextReward do
27             rewards[reward.name] += (stateCtr − 1, reward.value)
28         end
29     end
30     return model
31 end
32 function BUILDMODEL (method)
33     ast = PARSE (method)
34     model_commands = SYNTHESIS (ast)
35     model_preamble = ‘dtmc’+ ADDVARIABLES (condCtr) + ‘const int end_state
         =’ + stateCtr + ‘;\n’
36     model_preamble += ‘module’+ ast.methodName + ‘\n s : [0..end_state] init
         0; \n’
37     model_ending = ‘[] s=’ + stateCtr + ‘→ 1:(s′=’ + stateCtr + ‘);\n endmodule’ +
         ADDREWARDSTRUCTURES (rewards)
38     return model_preamble + model_commands + model_ending
39 end
```

missing, or in line 14 otherwise. In the latter case, the commands for the ‘else’ branch are then generated (line 15) and added to the *model* (line 16).

The *model* commands when *stmt* is a loop statement are generated in lines 19–22, by following a similar process to that used for a conditional statement, except that the last state modelling the loop body has its only outgoing transition leading back to the first state modelling the loop (line 22). To allow this, the *stateCtr* value for the first state of the loop commands is recorded in line 19.

Finally, when *stmt* is a return or an exception statement, a new *model* state is created (line 24). The only outgoing transition of this state points to the *end_state* of the *model*. This state is declared in the *model_preamble* in line 35 of BUILDMODEL and is generated in the *model_ending* in line 37 of BUILDMODEL, after the execution of SYNTHESIS finishes and the index of this state is known.

To enable the generation of the reward structures for the *model*, SYNTHESIS records the reward annotations from all statements (lines 26–28) into the *rewards* dictionary initialised in line 1. The reward structures are then included in the *model_ending* by invoking the auxiliary function ADDREWARDSTRUCTURES in line 37 of BUILDMODEL. Finally, the auxiliary function ADDVARIABLES is invoked in line 35 of BUILDMODEL to create the variable declarations for all unknown transition probabilities generated by SYNTHESIS for conditional statements and loops. The format of the reward structures and variable declarations generated by the two auxiliary functions is illustrated in the following example.

Example 3.2. *Figure 3.5 shows the PRISM-encoded DTMC model generated by Algorithm 1 for the* `distance1` *Java method from our running example. The model has two reward structures, corresponding to the* `time` *and* `cost` *annotations from the Java code in Figure 3.1. The transition probabilities* `p1` *and* `p2` *correspond to the 'if' statement and 'while' loop from the Java code. Their values depend on the usage profile of the code, and are determined as described in the next section.*

The current version of our PROPER prototype tool supports the analysis of a subset of Java features and programming constructs, due to the constraints discussed earlier. The Backus-Naur Form (BNF) found under Section A.1 of the Appendix A provides a detailed description of the subset's syntax. This syntax can be easily expanded to include additional Java features or constructs. The main constraint of the prototype tool's current version is the handling of multiple Java methods within the same DTMC model. This can be addressed by creating a module for each function located inside the method of interest. The result would be a DTMC model containing multiple modules for each function that appears in the code, and whose contents can be associated with properties of interest. However, further experimentation is necessary in order to prove

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const int end_state = 8;
6
7  module distance1
8    s : [0..end_state] init 0;
9
10   [] s=0 -> p1:(s'=1)+(1-p1):(s'=2); //line:3
11   [] s=1 -> 1:(s'=end_state);        //line:4
12   [] s=2 -> 1:(s'=3);                //line:8
13   [] s=3 -> 1:(s'=4);                //line:9
14   [] s=4 -> p2:(s'=5)+(1-p2):(s'=7); //line:10
15   [] s=5 -> 1:(s'=6);                //line:11
16   [] s=6 -> 1:(s'=4);                //line:12
17   [] s=7 -> 1:(s'=end_state);        //line:14
18   [] s=8 -> 1:(s'=8);
19 endmodule
20
21 rewards "cost"
22   s=1 : 7;
23 endrewards
24
25 rewards "time"
26   s=5 : 2.5;
27 endrewards
```

Fig. 3.5: PRISM model synthesised for the `distance1` Java method

the correctness of this extension and is part of the future work as discussed later in Section 6.2.1 of the thesis.

3.2.2 Transition Probability Calculation

The transition probabilities for the DTMC states modelling conditional statements and loops are calculated from the usage profile of the analysed code. PROPER requires a usage profile that provides, for each conditional statement and loop, the (expected) number of executions of the 'if' branch of the conditional statement or of body of the loop, respectively, over N_0 executions of the analysed code. There are multiple ways in which this usage profile can be obtained:

- directly from the program logs, if the code is instrumented to log this information;

- through a technique called *model counting* [22], which can calculate expected values for these counts from empirical probability distributions of the program inputs, where these distributions are taken from program logs;
- by Monte Carlo simulation applied to a simplified version of the code, where the program inputs for the simulation are drawn randomly from logs that reflect the empirical probability distributions of these inputs. Regarding the validity of the simplification process, only the statements with no impact on the required execution counts are removed, i.e., statements that do not affect the code's operational profile. The code's simplification is due to the constraints discussed at the beginning of Section 3.2, following the syntax of the Java subset that our PROPER prototype tool can handle.

Given a usage profile with these characteristics, consider a set of $n \geq 1$ nested conditional statements and/or loops from the analysed code. If the execution counts for these conditional statements/loops are $N_1, N_2, \ldots, N_n$,[5] then the transition probability associated with the i-th conditional statement/loop is calculated as:

$$p_i = \begin{cases} \frac{N_i}{N_{i-1}}, & \text{if statement } i \text{ is a conditional} \\ \frac{N_i}{N_{i-1}+N_i}, & \text{otherwise (if statement } i \text{ is a loop)} \end{cases} \tag{3.2}$$

where $1 \leq i \leq n$. For conditional statements and loops that are not nested within other conditional statements/loops (such as those from our running example), the number of executions of the analysed code is used in (3.2), i.e., $N_{i-1} = N_0$. Note that the number of executions of the else branch of a conditional statement is denoted as N_i', and its value is obtained by subtracting N_i from the number of executions of the analysed code, i.e., $N_i' = N_0 - N_i$. For conditional statements that are nested within other conditional statements/loops N_0 in the previous formula is replaced by N_{i-1}.

Example 3.3. *Suppose that the usage profile for the Java method* `distance1` *from our running example indicates that across* $N_0 = 10,000$ *invocations of the method, the if branch of the conditional statement starting in line 3 from Figure 3.1 was executed* $N_1 = 15$ *times, and the body of the while loop from lines 10–13 was executed* $N_2 = 254,000$ *times. Accordingly, the values of the unspecified transition probabilities for the DTMC model from Figure 3.5 are given by* $p1 = \frac{N_1}{N_0} = \frac{15}{10,000} = 0.0015$ *and* $p2 =$

[5]For a conditional statement, the count is of the number of executions of the if branch, if this branch is part of the statement nest, or of the else branch, if this branch exists and is part of the statement nest. For a loop, the count is of the number of executions of the statements within the body of the loop.

$\frac{N_2}{N_1'+N_2} = \frac{254{,}000}{(10{,}000-15)+254{,}000} = 0.9621$, *where* N_1' *represents the number of executions of the else branch starting in line 7 of the Java method from our running example.*

The following result shows that the PROPER probabilistic model synthesised in Section 3.2.1 and instantiated with the probabilities calculated above can be used to determine the performance properties of the code under analysis.

Theorem 3.1. *Given a Java method annotated with a performance property* (3.1)*, its DTMC D generated by Algorithm 1, and the DTMC transition probabilities* (3.2) *calculated for a usage profile of the method, the expected value of a random variable associated with the property for this usage profile is given by the probabilistic model checking of the reward property* $R_{=?}[F\ s = end_state]$ *over D.*

Proof. The performance properties analysed by our PROPER method are *additive*, i.e., if the execution time, cost or resource use under analysis is due to multiple program statements, the analysis can be carried out by adding up the property values determined separately for each of these statements. As such, we only need to prove the theorem for a property that associates a value $v > 0$ with a single program statement. We consider the general case where this statement is part of the body of $n \geq 0$ nested loops and/or conditional statements. Given N_0 program executions representative for the analysed usage profile, let $N_i \geq 0$, $1 \leq i \leq n$, be the total number of executions of the n-th such loop/conditional statement over the N_0 program executions.

The relevant part of the DTMC model D generated for the analysed code (i.e., the part modelling the n loop/conditional statement nest) comprises (a) n nested loop/conditional statement model constructs with the structure from Figure 3.3 and probabilities $p_{\mathsf{while}} = p_1, p_2, \ldots, p_i$ given by (3.2); and (b) a reward structure that associates the value v with a state within the innermost of these constructs. As such, the probabilistic model checking of the reward property $R_{=?}[F\ s = end_state]$ over D yields the expected reward value:

$$r = f_1 f_2 \ldots f_n \cdot v, \tag{3.3}$$

where f_i is a multiplicative factor associated with the i-th model construct, $1 \leq i \leq n$. For a model construct associated with a loop, this factor is given by

$$\begin{aligned} f_i = p_i(1 + p_i(1 + p_i(\ldots))) &= \lim_{k\to\infty}\left(p_i \tfrac{1-p_i^k}{1-p_i}\right) \\ &= \tfrac{p_i}{1-p_i} = \frac{\frac{N_i}{N_{i-1}+N_i}}{1-\frac{N_i}{N_{i-1}+N_i}} = \frac{N_i}{N_{i-1}} \end{aligned} \tag{3.4}$$

```
1  public static double power (int n1,
   ↪  int n2) {
2      while (n1 < 5) {
3          n1 = n1+1;
4          while (n2 < 5 ) {
5              n2=n2+1;
6          }
7      }
8      return Math.pow(n1,n2);
9  }
```

(a) Java method `power`

```
1  dtmc
2  const double p1; const double p2;
3  module power
4   s : [0..4] init 0;
5   [] s=0 -> p1:(s'=1)+(1-p1):(s'=4);//1:2
6   [] s=1 -> 1:(s'=2);               //1:3
7   [] s=2 -> p2:(s'=3)+(1-p2):(s'=0);//1:4
8   [] s=3 -> 1:(s'=2);               //1:5
9   [] s=4 -> 1:(s'=4);               //1:8
10 endmodule
11 rewards "time"
12  s=3 : 1;
13 endrewards
```

(b) Synthesised DTMC model

Fig. 3.6: Java method `power` on the left side of the figure, and its synthesised PRISM model representation on the right side.

due to the repeated execution of i-th loop with probability p_i. For a model construct associated with a conditional statement, the factor is simply $f_i = p_i = \frac{N_i}{N_{i-1}}$. Replacing these factor values in (3.3) gives an expected reward value

$$r = \frac{N_1}{N_0} \cdot \frac{N_2}{N_1} \cdot \ldots \cdot \frac{N_n}{N_{n-1}} \cdot v = \frac{N_n}{N_0} \cdot v, \tag{3.5}$$

i.e., the mean value of the analysed property for the considered usage profile (because the value v is associated with a statement executed N_i times across N_0 program executions). □

The above theorem proves that PROPER can obtain the value of additive reward properties. Non-additive or reachability properties, such as the probability of reaching a state n in the model ($P_{=?}[F\ s = n]$), could also be used to formalise reliability requirements of the code. The current version of our approach only partially supports this type of property. Specifically, we can obtain the probability of reaching a state inside a Java method if it does not contain loops; conditional statements are supported. The reason for this is that calculating the probability of reaching a state inside a loop is different than the probability of satisfying the loop's condition, which we already support in our transition probability calculation step. Preliminary experimentation has shown the need of expanding the current used models or constructing new ones to support the calculation of reachability properties.

Example 3.4. *We show this limitation using the simple Java method* `power` *from Figure 3.6a, whose associated parametric DTMC model is provided in Figure 3.6b. Consider now that we want to obtain the probability of reaching state $s = 3$ in Figure 3.6b. We start by calculating the values of the transition probabilities $p1$ and $p2$*

using the method's usage profile. Across $N_0 = 10,000$ *invocations of the method, the first while loop starting in line* 2 *of Figure 3.6a was executed* $N_1 = 13,457$ *times and the second while loop from line* 4 *was executed* $N_2 = 18,375$ *times. Accordingly, the values of the transition probabilities are given by* $p1 = \frac{N_1}{N_0+N_1} = \frac{13{,}457}{10{,}000+13{,}457} = 0.5736$ *and* $p2 = \frac{N_2}{N_1+N_2} = \frac{18{,}375}{13{,}457+18{,}375} = 0.5772$*. The analysis of* $R = ?[F\ done]$ *yields* $time = 1.836$*, as expected. However, the analysis of* $P = ?[F\ s = 3]$ *yields 0.437, which differs from the actual value of* 0.204 *for the probability of reaching the associated line from the Java method.*

3.2.3 Probabilistic Model Checking

In this PROPER step, we use a probabilistic model checker, e.g., PRISM [109] or Storm [60], to analyse the PCTL-encoded performance properties of interest over the DTMC synthesised by Algorithm 1, with the probabilities computed in (3.2).

Example 3.5. *Consider again our running example (Section 3.1). Determining the values of the 'cost' and 'time' properties specified using PROPER annotations in Figure 3.1 involves the probabilistic model checking of the reward PCTL properties* $R\{\text{"cost"}\} = ?[F\ s = end_state]$ *and* $R\{\text{"time"}\} = ?[F\ s = end_state]$ *over the DTMC model from Figure 3.5. To carry out these analyses for the usage profile from Example 3, the unspecified DTMC probabilities need to be initialised such that* $p1 = 0.0015$ *and* $p2 = 0.9651$*. The results of these analyses (using PRISM) are* $cost = 0.0105$ *and* $time = 69.0275$*.*

3.2.4 Further Application Scenarios

Besides supporting the analysis of the performance properties specified by the initial code annotations, the PROPER DTMC model can be reused for additional analyses in scenarios encountered in software engineering practice. One such scenario occurs when a method invocation from the analysed code is replaced with the invocation of a functionally equivalent method with different performance characteristics.

Example 3.6. *The impact of replacing the Math.abs function call from line 11 of the* `distance1` *Java method from Figure 3.1 with a call to the improved function FastMath.abs can be analysed using the same DTMC model as in Example 4, after only updating the reward value from line 25 of the model (see Figure 3.5) to match the specifications of the new function.*

Another scenario in which the DTMC model can be reused is when the code needs to be deployed on a new hardware platform with different quality attributes. As shown

by the following example, new quality properties can be analysed in this scenario by defining new reward structures for the DTMC.

Example 3.7. *Suppose that the application using the method* `distance1` *from our running example needs to be deployed on a smart phone on which its invocations of checkEqualLength and Math.abs consume 90 and 85 units of energy, respectively. The expected energy consumption of* `distance1` *can be predicted before actually running the application on the new hardware, by simply augmenting the DTMC model from Figure 3.5 with the new rewards structure*

```
rewards "energy"
  s=0 : 90;
  s=5 : 85;
endrewards
```

where $s = 0$ *and* $s = 5$ *are the DTMC states modelling the statements that use checkEqualLength and Math.abs.*

3.3 Implementation

To automate the performance analysis of probabilistic programs using PROPER, we implemented a tool with the architecture in Fig. 3.2. Our PROPER tool uses JavaParser[6] to parse the Java code of interest and generate the corresponding DTMC models (Section 3.2.1). We developed a customised Monte Carlo simulation method in Java to calculate the transition probabilities (Section 3.2.2) and employ the probabilistic model checker PRISM [109] to analyse properties of interest (Section 3.2.3). The PROPER open-source prototype tool, the full experimental results summarised next, additional information about our approach and the case studies used for its evaluation are available at https://github.com/is742/PROPER.

3.4 Evaluation

3.4.1 Research Questions

We evaluated PROPER by performing extensive experiments to answer the following research questions.

[6] https://javaparser.org

RQ1 (Accuracy): How accurately does PROPER support the analysis of non-functional properties of interest? We used this research question to establish if our method can achieve the same accuracy levels compared to the standard practice of analysing quality properties of interest via simulation or by running the system in normal working conditions.

RQ2 (Efficiency): What are the computational overheads of PROPER? We evaluated the execution time and memory footprint incurred by PROPER and compared them against the overheads incurred by simulation or real system execution.

3.4.2 Experimental Setup

We applied PROPER in multiple scenarios using Java source-code adapted from four Java libraries and applications:

1. The `distance1` Java method from the Apache Commons Math library[7] (see running example in Section 3.1).
2. The `getDevicePerfomanceClass` method from the Android messsaging app Telegram[8] (abbreviated '`devPerf`' in this section). Given a mobile device in which Telegram operates, this method identifies the specifications of the operating device and determines its performance class. The performance categories that a device can be linked with are: low, average and high. In our case study, we assumed that based on the result returned by this method, Telegram adapts to the specifications and shifts the performance of some of its features. Additionally, we introduced a new performance category (very high) to show the applicability of our approach in cases where additional code is being introduced.
3. The `fst` method from the Apache Commons Maths library. This method implements the fast sine transformer algorithm for one-dimensional real data sets.
4. The `knapsackDP` method from a public tutorial series on GitHub.[9] This method is an implementation of the widely used dynamic-programming knapsack algorithm.

Table 3.1 provides an overview of our case studies, along with a list of identified performance properties of interest, formally expressed in PCTL [85], that can be evaluated using our tool-supported PROPER method. The Java code for each of the

[7]https://commons.apache.org/proper/commons-math/

[8]https://github.com/DrKLO/Telegram/

[9]https://github.com/eugenp/tutorials/

Table 3.1: Description of case studies' models and properties of interest expressed in both natural language and PCTL.

Case studies	#states	#trans.	#linesOfcode	Performance property description	PCTL
distance1	8	10	16	What is the expected time?	$R\{\text{"}time\text{"}\} = ?[\,F\ s = end_state\,]$
				What is the expected cost?	$R\{\text{"}cost\text{"}\} = ?[\,F\ s = end_state\,]$
devPerf	17	21	40	What is the expected energy consumption?	$R\{\text{"}energy\text{"}\} = ?[\,F\ s = end_state\,]$
fst	30	35	47	What is the expected time?	$R\{\text{"}time\text{"}\} = ?[\,F\ s = end_state\,]$
				What is the expected cost?	$R\{\text{"}cost\text{"}\} = ?[\,F\ s = end_state\,]$
knapsackDP	18	23	29	What is the expected energy consumption?	$R\{\text{"}energy\text{"}\} = ?[\,F\ s = end_state\,]$
				What is the expected time?	$R\{\text{"}time\text{"}\} = ?[\,F\ s = end_state\,]$

above scenarios along with the respective DTMC models, generated by PROPER, can be found in Appendix A.

For the evaluation of all research questions, we assume that the values of the rewards of interest linked with a service or state, e.g., cost, execution time or energy consumption, are obtained from the service provider, and that logs capturing the program's usage profile are available. In the `distance1` case study, we measure the expected time and cost associated with the *Math.abs* method and with throwing the exception, respectively. In the `devPerf` case study, we are interested in the expected energy consumption of running the code, due to an *Animations* method that sets the level of the application's visual quality. Depending on its input mode, each instance of this method is linked to a different amount of energy (28, 34, 40 or 48). Similarly, in the `fst` case study, we measure the expected time associated with the *FastMath.sin* method (where each execution takes 1.5 time units), together with the expected cost of reaching any of the two exceptions (of cost 5 each). Finally, in the `knapsackDP` case study, we are interested in the expected energy consumption due to a *display* method located in the code (whose executions use 67 units of energy each), and in the expected time associated with the *Math.max* method, each invocation of which takes 2 time units.

All experiments were run on a macOS Big Sur Macbook Pro with 2 GHz Dual-Core Intel Core i5 CPU and 8 GB RAM. The source code, Markov models, data used for the experimental evaluation and full experimental results are publicly available in our GitHub repository https://github.com/is742/PROPER.

3.4.3 Results and Discussion

We answer RQ_1 by comparing the PROPER results with those produced by simulating the execution of the programs from Table 3.1 in a realistic environment and with a suitably instrumented operational profile. To achieve this, we deployed the code of each

Table 3.2: Comparison in accuracy of results obtained using PROPER and simulation.

Properties	PROPER				Simulation			
	distance1	fst	knapsackDP	devPerf	distance1	fst	knapsackDP	devPerf
R{"tim"}=?[F s=*end_state*]	2.5	1.14	21.96	N/A	2.5	1.11	21.19	N/A
R{"cost"}=?[F s=*end_state*]	4.66	1.91	N/A	N/A	4.63	1.89	N/A	N/A
R{"energy"}=?[F s=*end_state*]	N/A	N/A	735.93	30.96	N/A	N/A	710.24	31.02

program in a mobile device in the form of a stand-alone application using the Android studio's emulator and performed simulation directly on the device.

Table 3.2 shows the results obtained from the verification of properties of interest using PROPER and simulation. The N/A option in the table refers to the property not being applicable to that specific case study; thus, no value has been reported. To execute the PMC step of PROPER (Section 3.2.3), we used the PRISM model checker [109] and provided as input to the DTMC models the probabilities obtained during the transition probability calculation step of the approach (Section 3.2.2). Then, we quantified the properties shown in Table 3.1. To obtain the values from simulation, we performed 10^4 simulated runs of each case study. The input for the methods during every simulation was randomly selected from the program's log. Due to the randomness of selecting data from the log, the results, as expected, were slightly different every time we performed the simulation. To alleviate this validity threat (cf. Section 3.4.4) and to increase the accuracy of simulation results, we created 10 sets of simulated runs of 10^4 code executions and calculated the average property values.

As can be seen from the results in Table 3.2, the quality properties evaluated using PROPER are within 3.5% of those obtained in simulation. The small differences in the results in Table 3.2 are due to the randomness in simulation. Increasing the number of simulation runs would reduce further the delta; experimenting further on this research thread is part of our future work.

Summary for RQ_1: These above results confirm the capability of our approach to accurately analyse performance properties of probabilistic programs without the need to execute the source code in simulation.

To answer RQ2, we measured the execution time and memory consumption of running the code in real time with obtaining results using PROPER. To measure the code's execution time we used the *currentTimeMillis* method from Oracle's *System*(https://docs.oracle.com/javase) class and for the probabilistic model checking step of PROPER we used the output log from PRISM [109] to obtain both the time needed for model construction and model checking for each of the specified properties. We

Table 3.3: Time and memory consumption comparison between PROPER and simulation.

Properties	PROPER				Simulation			
	distance1	fst	knapsackDP	devPerf	distance1	fst	knapsackDP	devPerf
Execution time (seconds)	0.003	0.005	0.01	0.004	256.2	193.2	2826.6	264.6
Memory consumption (MB)	[12-39]	[12-36]	[12-37]	[11-36]	[3.4-37.7]	[4.3-37.4]	[6-49]	[2.7-36]

measured the memory consumption using the *JavaVisualVM* profiling tool which comes with the Java Development Kit (JDK). Also, we used the method *sleep* from Oracle's *Thread*class to simulate a server response time of 2ms for each function invocation.

The experimental results in Table 3.3 show that PROPER is much faster than executing the code in its operating environment. In terms of memory, PROPER independent of the case study consumes on average the same amount of memory. With simulation, however, the `knapsackDP` method which had a longer execution time than the rest case studies, showed an increase in the min and max values of used memory too.

Summary for RQ_2: The above results reveal PROPER method's efficiency compared against the overheads of simulation or real execution, both in terms of execution time and memory consumption.

Exploitation of PROPER in Engineering Scenarios

PROPER can be a useful tool in helping software engineers making informed decisions in various engineering scenarios. Below we present two modification scenarios (*Scenario A* and *Scenario B*) that frequently occur in the domains of product obsolescence [14] and software modernisation [56].

In *Scenario A*, software engineers replace one of the external methods used by the program of interest to optimise the requirements defined during the design phase of the program. Such a modification may involve, for example, replacing an existing external method with a faster alternative to reduce response time, or using a less reliable but cheaper method to reduce the operational cost, provided that the method does not critically affect the application's functionality. Since the operational profile of the application does not change, and given the reward values for the new method by the service providers in the form of a service-level agreement, we can use PROPER to quantify quality properties of interest without simulating the code's execution. This will not only save time and effort, but it will also enable engineers to verify additional properties that were not considered during system design.

Table 3.4: Results obtained using PROPER for two different scenarios. Scenario A: replacement of a program method with a functionally-equivalent method with different performance characteristics. Scenario B: Program deployment on a new hardware platform with different quality attributes.

Properties	Scenario A				Scenario B			
	distance1	fst	knapsackDP	devPerf	distance1	fst	knapsackDP	devPerf
R{"time"}=?[F s=*end_state*]	**1.8**	**1.67**	**14.27**	N/A	**3.2**	**1.97**	**30.75**	N/A
R{"cost"}=?[F s=*end_state*]	4.66	1.91	N/A	N/A	4.66	1.91	N/A	N/A
R{"energy"}=?[F s=*end_state*]	N/A	N/A	**856.76**	**26.27**	N/A	N/A	**900.69**	**35.73**

Table 3.4 shows the updated results in bold obtained during *Scenario A* for the selected case studies. In `distance1` case study, we used the method *FastMath.abs* that offered improved execution time (=1.8) instead of *Math.abs* whose execution time was 2.5. The expected cost was not affected by this change, as it is associated with the exception. In the `fst` case study, we replaced the *FastMath.sin* method with the slower (=2.2 per invocation) but more reliable *Math.sin* method which resulted in a slight increase in execution time (i.e., 1.14 with *FastMath.sin* vs 1.67 with *Math.sin*). Similarly to `distance1`, the cost was not affected. The change in the `knapsackDP` program affected both the expected time and energy consumption. In particular, we introduced the faster (=1.3) method *FastMath.max* instead of the *Math.max* method, which resulted in reduced execution time (14.27 vs 21.96). We also updated the *display* method to increase performance using a more computationally-expensive method (=78), which led to increased overall energy consumption (735.03 vs 856.76 before and after the change, respectively). Finally, in the `devPerf` program we assumed that the *Animations* method was updated to offer better optimisations making use of the increased number of cores in modern mobile devices (=23,30,35,43). This change resulted in a decrease of energy consumption (30.96 vs 26.27) in all its invocations.

In *Scenario B*, software engineers do not make any internal changes in the code; instead, the application is deployed in a new device with different capabilities and specifications. Such scenarios may arise when transferring the same software between mobile devices or when deploying the same software in robotic systems with different performance, memory, networking and other characteristics (e.g., a robot using a Raspberry Pi 4 and another using a Raspberry Pi Zero).

Since the applied changes are only external and the operational profile of the application does not change, we can employ PROPER and obtain the updated values for the quality attributes of interest. Table 3.4 (*Scenario B*) shows in bold the updated values of the performance properties for the four applications assuming that they have been deployed in a device with reduced hardware performance.

The experimental results from both scenarios show that PROPER can provide useful insights on the impact of potential internal changes in the code or external in the operating environment of an application. The impact of such changes can be assessed without updating the code or deploying it in the target hardware platform, thus reducing significantly the effort and cost in analysing performance properties of interest. These results provide evidence how PROPER can assist software engineers in making informed decisions.

3.4.4 Threats to Validity

Construct validity threats may arise from the construction of the case studies' models based on the selected Java code. To mitigate this threat, all use cases are based on real-world applications, and the produced models refer to parts of these applications' source code. The presented case studies are selected from various domains to cover several types of Java applications, including mobile and desktop computing. While the selected case studies are indicative of the generality of the approach, more experiments are necessary for PROPER's future extensions. For instance, such extensions are needed to cover cases with additional programming constructs and Java features.

Internal validity threats can originate from obtaining inaccurate results via simulating the code's execution. To mitigate these threats, we performed simulation up to 10^4 times. Additionally, we created 10 sets of these simulation runs and calculated the average of their output values.

External validity threats might be due to the difficulty of representing part of a Java application's source code as a DTMC model. To mitigate this threat, we carefully compared each model to its respective code method, and built an automated implementation of PROPER to assist us in the code-to-model transformation process. As previously stated, further experiments are needed to evaluate our method for additional code samples.

3.5 Related Work

Probabilistic software analysis (PSA) [62] has been used successfully in many domains, including testing, cryptographic protocols, cyber-physical systems, biology, and reliability analysis [83]. However, to the best of our knowledge, our method is the first PSA approach that synthesises a probabilistic model (DTMC) directly from source code to verify properties of interest that appear within the code. The only related work we are aware of belongs to the areas of *software maintenance* [17] and *software reliability*

analysis [116]. Unlike our approach, research in these areas uses mostly techniques such as symbolic execution [73, 164] and simulation [80, 88], rather than probabilistic model checking.

Probabilistic symbolic execution, firstly introduced in [73], is an extension of symbolic execution that allows probabilistic reasoning. It has been implemented as an extension of the Symbolic PathFinder system to perform quantitative reasoning, i.e., the calculation of path probabilities within a program. In order to calculate these probabilities, model counting techniques [81, 82, 166] have been successfully applied. Based on this work, the authors of [164] describe a probabilistic environment for Java that is based on symbolic execution. This framework can handle probabilistic programming features, such as observe statements, and can be used for the encoding and analysis of Discrete Time Markov Chains (DTMC), Bayesian Networks, etc. The main difference between the listed approaches and ours lies in the way that symbolic execution deals with loop constructs. While a bound is required for the exploration depth, our approach achieves precise exploration of loops.

Additionally, the approach presented in [69] introduces a general methodology that uses symbolic execution of source code for extracting failure and success paths that can be used for probabilistic reliability assessment, against relevant usage scenarios. The result of symbolic execution is a finite set of paths, each with a path condition. These paths can either lead to success, failure or can be interrupted by the bounded exploration. This approach performs reliability analysis directly on source code, in contrast with most of the current approaches that are limited on architectural level. However, only reliability has been addressed, and the bounded exploration can potentially lead to loss of information necessary for nonfunctional property analysis. Our research aims to address the problem of bounded exploration of loops, and to consider additional nonfunctional properties, e.g. performance. The research in [70] extends the previous approach, by building upon the symbolic execution framework, with the aim of computing a precise numeric characterization of program changes. This leads to the ranking of different program changes based on their probability of execution and their impact on target events.

The approach introduced in [171] performs reliability assessment by using software metrics for reliability modeling which are collected from source codes of post versions. During the application of the approach, redundant metric elements are filtered out and the rest are aggregated to represent the module reliability. Moreover, the authors propose a framework to automatically apply the module value and calculate overall reliability by introducing formal methods. This work differs from ours, as it uses DTMC models

built around the control transfer relationship between components and it is not directly applied on source code. Also, its main focus is only towards reliability assessment in comparison with our approach which targets multiple nonfunctional properties.

Focusing on simulation based approaches, the project described in [80] develops simulation procedures to assess the impact of individual components on the reliability of an application in the presence of fault detection and repair strategies that may be employed during testing. Additional simulation procedures are also developed to analyze the application reliability for various operational configurations. While this approach and ours are similar in the sense that they both achieve precise exploration of loops, they have clear differences related to the nature of the used verification method (simulation and probabilistic model checking). Probabilistic model checking outperforms simulation, which can also be seen in our evaluation, and it can also be used to verify properties of interest identified at later stages of software development without the need of re-running the code.

On a different track, [61] introduces two automated reduction methods for probabilistic programs that operate purely on a syntactic level. The aim of this research is to address the need of reduction methods to automatically generate probabilistic models for reliability analysis, which turned out to either being not feasible due to memory constraints or where the reliability analysis exceeds reasonable time limits. The approach proposed in [88] implements an adaptive framework of incorporating path testing into reliability estimation for modular software systems. Three estimated methods based on common program structures (sequence, branch, loop) are proposed to calculate the path reliability. The derived path reliabilities can be applied to the estimates of software reliability. Both of these approaches differ from ours as they focus on the verification of reliability related properties and use bounded exploration of loops, whereas our approach focuses on model generation from source code with the ability to verify various nonfunctional properties and deal with bounded exploration of loops.

The nonfunctional property analysis techniques are not limited at code-level, with many existing approaches focusing at system architecture-level. The following research work belongs on the second category. An overview of all listed approaches in this section can be seen in Table 3.5.

Table 3.5: Overview of related research

Approach	Type of analysis	Input	Models	Properties	Loop handling	Tool support
Architecture-based approaches						
Synthesis and quantitative verification [35]	Probabilistic model checking	Architectural style and elements	DTMCs	Reliability, performance, cost	N/A	N/A
Improving self-adaptation planning [71]	Probabilistic model checking	Software architecture in the form of an ADL	DTMCs	Reliability	N/A	N/A
Exploiting traceability uncertainty [160]	Mean value analysis (MVA) and simulation	Annotated software architectural model	Queueing Network (QN) models	Performance, security	N/A	Yes
Model-based verification of quantitative properties [79]	Probabilistic model checking	Variable sequence diagrams (SDs)	DTMCs	Reliability, energy consumption	N/A	Yes
Performance and reliability analysis [49]	COBRA [140] and PRIMA [47] methodologies	UML diagrams (use case, sequence, deployment)	Model driven architecture (MDA) models	Reliability, performance	N/A	N/A
Code-based approaches						
Quantification of software changes [70]	Model counting, symbolic execution	Java code	Symbolic execution tree	Reliability	Bounded exploration depths for loop structures	Symbolic PathFinder (SPF)
Probabilistic programming for Java [164]	Model counting, symbolic execution	Java code	Symbolic execution tree	Reliability	Bounded exploration depths for loop structures	Probabilistic extension of symbolic PathFinder (SPF)
Reliability analysis in symbolic Pathfinder [69]	Model counting, symbolic execution	Java code	Symbolic execution tree	Reliability	Bounded exploration depths for loop structures	Symbolic PathFinder (SPF)
Probabilistic symbolic execution [73]	Model counting, symbolic execution	Java code	Symbolic execution tree	Reliability	Bounded exploration depths for loop structures	Probabilistic extension of symbolic PathFinder (SPF)
Reduction methods [61]	Probabilistic model checking	SIMULINK models	DTMCs	Reliability	Bounded exploration depths for loop structures	Yes
Structure-based software reliability model [171]	Reliability calculation based on module parameters and expression forms	Control transfer relationship between components	DTMCs	Reliability	N/A	Yes
Structure-based software reliability analysis [80]	Simulation	Methods in C programming language	Nonhomogeneous continuous-time Markov chains (NHCTMCs)	Reliability	Precise exploration of loops	N/A
An Adaptive Reliability Analysis [88]	Simulation, path reliability calculation	Control flow graph	Markov processes	Reliability, Criticality	Bounded exploration depths for loop structures	N/A
Our approach						
Probabilistic analysis of code performance	Probabilistic model checking	Java code	DTMCs/ pDTMCs	Performance, cost, energy consumption	Precise exploration of loops	Yes

The approach proposed in [35] combines synthesis of spaces of system design alternatives from formal specifications of architectural styles with probabilistic formal

verification. Additionally, the work presented in [71] introduces an improvement to the planning stage of self-adaptive systems by predicting the outcome of each adaptation strategy. Specifically, a stochastic model is derived from a formal architecture description of the managed system with the changes imposed by each strategy. This information is then used to optimize the self-adaptation decisions to fulfill the desired quality goals.

The approach introduced in [160] aims to automate the traceability between software architectural models and extra-functional results by investigating the uncertainty while bridging these two domains. This approach makes use of extra-functional patterns and antipatterns, to deduce the logical consequences between the architectural elements and analysis results. By building a graph of traces, it becomes possible to identify the most critical causes of extra-functional flaws. The model-based approach described in [79] enables software engineers to assess their design solutions for software product lines in the early stages of development. A nonfunctional MDA framework (NFMDA) is considered in [49]; it embeds new types of model transformations that allow the generation of quantitative models for nonfunctional analysis. By using the framework with two methodologies, one for performance analysis and one for reliability assessment, an illustration of the relationships between nonfunctional models and software models is achieved.

The approaches mentioned above differ from ours as they target the software system at the architectural level and do not provide insights on the nonfunctional properties existing at code-level. However, they include a wider variety of verifiable properties in contrast to the code-level approaches, which are mainly focused towards reliability. Our approach is similar to the architecture-based approaches in terms of variety of identified properties.

3.6 Summary

We presented PROPER, a tool-supported method for the formal analysis of performance properties of the components of a software system. PROPER synthesises a DTMC using code annotated with performance properties of interest (e.g., timing, resource use, cost), calculates the transition probabilities of the DTMC using program logs, and executes probabilistic model checking to quantify these properties. We evaluated PROPER on four applications and demonstrated how it can support the performance analysis in scenarios involving changes in hardware platforms, function libraries or usage profile.

Chapter 4

Software System Analysis and Refinement Using Performance Antipattern Profiles

Over the last two decades, research highlighted the importance of integrating quantitative analysis in the software development process, in order to ensure that nonfunctional requirements have been met [50]. Prominent techniques in this area, such as performance antipatterns [144] and probabilistic model checking [95], have been extensively applied to enable the analysis of performance properties of software systems, and to support refactoring when requirements are violated.

While approaches enabling this type of analysis do exist [6, 26, 74], they are typically applicable only in cases where the SUD operational profile is known and does not exhibit changes over time. However, many of today's systems operate under uncertainty, often in the safety-critical domain. Thus, it is necessary to ensure that the system can continue to achieve its performance objectives and to identify any possible operational space areas where this might not be the case.

In this chapter, we introduce a novel performance analysis and refactoring approach that addresses this need. The new approach considers the uncertainty in the operational profile of a system under development by identifying the performance antipatterns present in predefined *operational profile regions*. These regions capture aleatoric and epistemic operational profile uncertainties due to unavoidable changes in the environment (e.g., workload variations) and to insufficiently measured environment properties (e.g., CPU speed), respectively.

A few existing solutions [2, 31, 66] employ sensitivity analysis to assess the robustness of software to variations in its operational profile. However, these solutions are incapable of capturing major operational profile changes like our approach, and therefore focus on establishing the effect of small operational profile variations on the performance of the SUD. In contrast, our new approach provides a global perspective on the performance antipatterns associated with a wide range of operational profiles. This perspective enables software engineers to identify operational profile regions in which their SUD is likely to require refactoring, and supports the selection of suitable refactoring actions for such regions.

The main contributions of the approach presented in this chapter are:

1. We introduce the concept of a *performance antipattern profile* (i.e., a "map" showing the antipatterns present in different regions from the operational profile space of a SUD), and a method for synthesising such profiles for systems comprising a mix of internal and external software components. We define as *internal* the components of a system that are deployed on the servers of the organisation or company that owns the system. *External* components are the system's components that are supplied by third-party providers, and therefore cannot be modified; they can only be replaced with other components that provide the same functionality.

2. We present a tool-supported approach that uses our performance antipattern profile synthesis method, and we define best practices for refactoring the architecture of a SUD using performance antipattern profiles.

3. We demonstrate the application of our approach for a software system comprising a combination of internal (i.e., in-house) components and external (i.e., third-party) services.

The remainder of this chapter is organized as follows. Section 4.1 introduces a software system that we use to illustrate the application of our approach throughout this chapter. Section 4.2 presents the new approach for the performance analysis and refactoring of software systems, and Section 4.3 provides a description of the approach's implementation to support the automated synthesis of antipattern profiles. Section 4.4 describes its application to the service-based system from our motivating example. Section 4.5 compares our solution with existing approaches. Finally, Section 4.6 summarises the benefits and limitations of our approach, and suggests directions for future work.

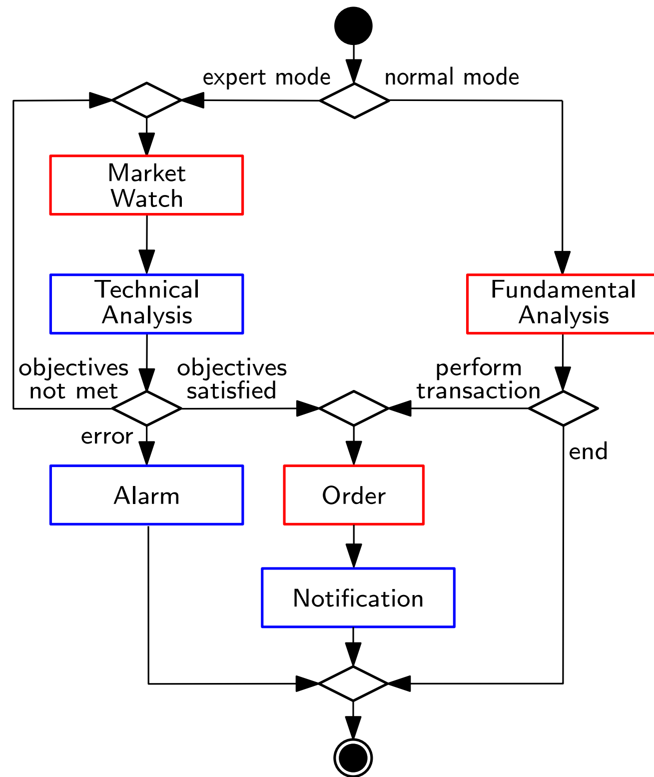


Fig. 4.1: Workflow of the foreign currency trading system (FX). The internal components of the system are depicted in blue colour and the external services in red.

4.1 Motivating Example

To illustrate the application of our approach, we consider a heterogeneous software system comprising both internal components and external services. We assume that the internal components are deployed on the private servers of the organisation that owns the system. As such, the architecture and resources of these components can be modified if needed. In contrast, the external services are accessed remotely from third-party providers and cannot be modified. These services can only be replaced with (or can be used alongside) other services that are functionally equivalent but may induce different performance.

4.1.1 System Description

The system we use as a running example is adapted from [34, 75] and comes from the foreign exchange trading domain. The workflow implemented by this "FX" system is shown in Figure 4.1, and involves handling requests sent by currency traders.

The user of the FX system (also known as trader) can choose between two operation modes. In the so-called "expert" mode, a loop is executed by the system that enables

the analysis of market activity, which identifies patterns that satisfy the objectives set by the trader. If the objectives are met, the system automatically completes the respective trade(s). In more detail, the "market watch" service extracts real-time exchange rates of selected currency pairs. The data obtained from "market watch" is then supplied to a "technical analysis" service that assesses the current trading conditions, considers the possible scenarios of future price movements, and determines whether the objectives set are (a) satisfied; (b) not satisfied; or (c) not satisfied with high variance. In the first case, an "order" service is invoked to finalise the trade. In the second case, "market watch" is re-invoked, and in the last case, an "alarm" service is invoked to inform the trader about errors or opportunities not covered by the current objectives. In the "normal" mode, FX evaluates the profile of a country from an economic perspective using a "fundamental analysis" service that enables the collection and analysis of information such as economic data and news reports, and concludes to an evaluation of the country's currency. The outcome of this assessment can also lead to the "order" service's invocation to buy or sell currency initiated by the trader, and a "notification" service confirms the trade's completion.

In line with the system's description, two types of requests are possible: requests that must be handled in "expert" mode, and requests handled in "normal" mode. The request type determines whether the system starts with a "fundamental analysis" (FA) operation or a "market watch" (MW) operation. Both of these operations use external services. "Technical analysis" (TA) is an operation provided by an internal component. As mentioned above, TA's assessment could result in case (c), i.e., an internal "alarm" operation is triggered to inform the user about this outcome. The optimal results of either technical or fundamental analysis (satisfied objectives/trade acceptance) lead to the execution of an external "order" operation that completes the trade, and is followed by an internal "notification" operation that confirms the successful completion of the workflow.

4.1.2 External Services

For the operations executed using external services, multiple services can be used as equivalent alternatives or in some combination deemed suitable. Given $n > 1$ functionally equivalent services, three options for combining them are possible:

- *Sequential* (SEQ): first invoke service 1; if the invocation succeeds, use its response; if it fails, then invoke service 2, etc., until service n is invoked, if needed. This option improves reliability, but it does not benefit performance.

- *Parallel* (PAR): invoke all *n* services at once, and use the first result that comes back. This option improves both reliability and performance, but it is expensive.
- *Probabilistic* (PROB): invoke one of the *n* available services, selected based on a discrete probability distribution. This option improves the average reliability and response time by invoking an expensive service once in a while.

Therefore, we need to choose a "good" option (i.e., one that enables the system to satisfy its performance requirements) starting from information about the performance characteristics shown by each of these services, which we assume known from either the service-level agreement (SLA) published by the providers of these services, from our observations, or from both. Additionally, we assume that all these services already satisfy the functional requirements.

4.1.3 Internal Components

The internal operations are executed by software components belonging to the organisation that "owns" the system, and are running on their private physical servers. We assume that technical analysis (TA) has a much more significant impact on the performance of the system compared to the other two in-house components (alarm and notification), which require only modest resources. Consequently, it is necessary to identify possible antipattern-driven refactoring actions for the TA component, to ensure that the system operates with an optimal performance. If and when needed, the refactoring actions we consider are: (i) duplicate the TA software component and load balance the incoming requests among the two TA instances; or (ii) replace the TA instance with a faster one. These actions will increase the cost, but may be needed to satisfy the performance requirements of the system.

4.1.4 Operational Profile Parameters

Several parameters of the system are outside the control of its developers. These parameters represent the *operational profile* of the system. For our FX system, they include the probability that a user request needs expert-mode handling, and the probability of a transaction being performed after the execution of the fundamental analysis operation (cf. Figure 4.1). The choice of these parameter ranges reflects, for instance, the engineers' expectation about a particular deployment of the system, numerical values will be provided in Section 4.4.

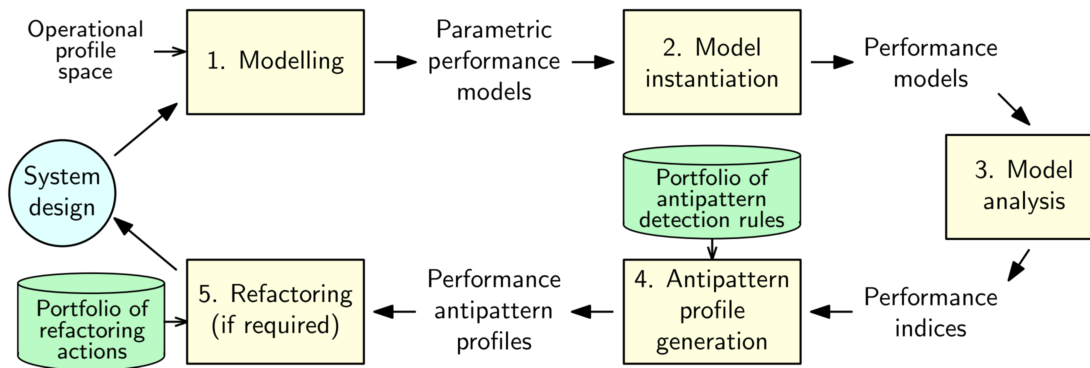


Fig. 4.2: Performance analysis and refactoring using antipattern profiles

4.2 Approach

As shown in Figure 4.2, our approach to performance analysis and system refactoring comprises five steps. In step 1, a model representation of the system of interest is developed with an emphasis on its performance characteristics. This step needs to be performed by an engineer with expertise in performance modelling, and its output is a set of parametric performance models which are instantiated, in step 2, for combinations of parameter values covering the system's entire operational profile space. The instantiated performance models are then analysed in step 3 to establish the performance indices related to all considered combinations of operational profile parameter values. In step 4, the performance indices along with a portfolio of antipattern detection rules are used to identify the occurrence of performance antipatterns for different combinations of parameter values. Finally, step 5 determines the need for refactoring actions, based on whether performance antipatterns are present or not in areas of the operational profile space where the system is expected to operate.

4.2.1 Modelling

Starting from an initial system design proposed by a software engineer, step 1 involves modelling the performance characteristics of the system across its entire operational profile space (i.e., for all possible values of the operational profile parameters). As such, the performance models produced by the modelling step are *parametric models*—models containing (uninstantiated) parameters like the probabilities of receiving different types of user requests. Our approach is not prescriptive about the type of performance models that can be used in its modelling step. However, these models must be able to capture the uncertainty associated with the operational profile of the system. Therefore, in this approach we use parametric discrete-time and continuous-time Markov chains (parametric DTMCs and CTMCs).

```
ctmc

//Parameters of the FX model
const double pObjNotMet;
const double pObjSatisfied;
const double pObjNotMetHighVar = 1-pObjSatisfied-pObjNotMet;
const double pExpertMode;
const double pFAMode = 1-pExpertMode;
const double pPerformTransaction;
const double MWsucc;
const double FASucc;
const double OrderSucc;
const double MWrate;
const double FARate;
const double OrderRate;
const double reqRate;
const int nthreads;
const int MAX_QUEUE_SIZE;
const double ta1Rate;
const double alarmRate;
const double notifRate;
const double internalOpRate;
```

Fig. 4.3: Parameters of the parametric CTMC model (FX)

In the case of our example, we construct parametric DTMCs to model the external services used by the FX system (Figure 4.1), namely market watch, fundamental analysis and order, and parametric CTMCs to model the system in its entirety.

Figures 4.3–4.7 depict a parametric CTMC modelling the system's (initial) architecture. As seen in Figure 4.3, the parametric CTMC model contains a set of parameters that are later instantiated in step 2 of the approach (Section 4.2.2) and correspond to the probabilities of following a path within the system (i.e., *pObjNotMet*, *pObjSatisfied*, *pObjNotMetHighVar*, *pExpertMode*, *pFAMode*, *pPerformTransaction*), the service rates of each of the internal components and external services (i.e, *MWrate*, *FARate*, *OrderRate*, *ta1Rate*, *alarmRate*, *notifRate*), the invocation probabilities for each of the external services (i.e, *MWsucc*, *FASucc*, *OrderSucc*), the arrival rate of incoming requests (i.e., *reqRate*), the number of system threads (i.e, *nthreads*), the service rate for internal operations (i.e, *internalOpRate*), and the size of the queue handling incoming requests (i.e, *MAX_QUEUE_SIZE*).

The model consists of a *RequestQueue* module (Figure 4.4) that handles the incoming requests, a main module *Workflow1* (Figure 4.5) that performs the FX system's main functionalities, and a *TA1* instance module (Figure 4.6) that, as an instance of the TA internal component, is responsible for executing TA's functionalities when

```
module RequestQueue
  q : [0..MAX_QUEUE_SIZE] init 0;

  [NewReq] true -> reqRate:(q'=min(q+1,MAX_QUEUE_SIZE)); //req
    arrival: increase req queue size (or drop request)
  [ServeReq1] q>0 -> internalOpRate:(q'=q-1); //thread 1 serves
    request
endmodule
```

Fig. 4.4: Request queue of the parametric CTMC model (FX)

invoked. Note that the number '1' following the names of the modules *Workflow1* and *TA1* indicates the thread and instance number, respectively. In our experimentation in Section 4.4, we cover cases where models are constructed with multiple threads and TA instances. An example of such model can be seen in Section B.2 of Appendix B.

The *Workflow1* module (Figure 4.5) models the main workflow of the FX system as follows. Starting from the initial stage, a request is extracted from the queue (Figure 4.4) and its type is established, namely expert or normal mode. In the former case, the MW external service is invoked with a service rate and a probability of success. If the MW invocation succeeds, an instance of the TA component is then invoked if it is not already in use (Figure 4.6). While waiting for the execution of the TA instance to be completed, its outcome will determine the next steps of the workflow. If the objectives set are satisfied, then the order external service is invoked to proceed with the trade. Following the trade's successful completion, the notification internal component notifies the user with an appropriate message. If the objectives are not satisfied, MW is re-invoked and the process is repeated until a trade that satisfies the objectives has been reached. There is also a case in which the objectives are not satisfied but with a high variance. In this case, the alarm internal component is triggered and informs the user about this variance from the objectives set. This operation concludes the different cases within the expert mode branch. In the latter case of normal mode, the FA operation is initiated with the invocation of associated external services. This operation can also lead to a potential trade which, if successful, initiates the invocation of both the order external service and the notification internal component.

The reward structures defined in the CTMC model are related to properties of interest and associated with the system's requirements. For example, the information we can obtain from the verification of the rewards as depicted in Figure 4.7, within a specified number of T time-steps, is the number of dropped requests (i.e, *droppedRequests*), the number of handled requests (i.e., *numOfReqsHandled*), the number of invocations for each of the internal components and external services (i.e., *MWcount*, *Notifcount*,

```
module Workflow1
  s1 : [0..9] init 0;

  //1. Extract request from queue & establish request type
  [ServeReq1] s1=0 -> pExpertMode:(s1'=1) + pFAMode:(s1'=8);

  //2.1. EM request - invoke ext. service(s) for the MW operation
  [] s1=1 -> MWsucc*MWrate:(s1'=2) + (1-MWsucc)*MWrate:(s1'=9);

  //2.2. Invoke internal component for the TA operation
  [TA1Invoke1] s1=2 -> 1:(s1'=3); // invoke TA instance 1
  [TA1Exec1]   s1=3 -> pObjSatisfied:(s1'=6) + pObjNotMet:(s1'=1) +
   pObjNotMetHighVar:(s1'=4); // wait for TA to complete

  //2.3. Invoke internal component for the Alarm operation
  [] s1=4 -> alarmRate:(s1'=5);

  //2.4. Done - successful outcome
  [] s1=5 -> internalOpRate:(s1'=0);

  //2.5. Invoke external service(s) for the Order operation
  [] s1=6 -> OrderSucc*OrderRate:(s1'=7) + (1-OrderSucc)*OrderRate:(
   s1'=9);

  //2.6. Invoke internal component for the notification operation
  [] s1=7 -> notifRate:(s1'=5);

  //3. FA request - invoke ext. service(s) for the FA operation
  [] s1=8 -> FASucc*FARate*pPerformTransaction:(s1'=6) + FASucc*
   FARate*(1-pPerformTransaction):(s1'=5)+(1-FASucc)*FARate:(s1'=9);

  //4. Done - unsuccessful outcome (ext.-service invocation failed)
  [] s1=9 -> internalOpRate:(s1'=0);
endmodule
```

Fig. 4.5: Main workflow of the parametric CTMC model (FX)

Alarmcount, *FAcount*, *Ordercount*, *servedTA*), the processing time of the system (i.e., *processingTime*), the number of failed invocations for the external services of the system (i.e, *extFails*), the number of served requests (i.e., *served*), the length of the queue (i.e., *qLen*), and the time required for invoking and executing the TA component (i.e., *taTime*).

The information available about the external components of a system is usually very limited, e.g., it is typically not possible to know how a third-party web service is organising its request buffering, how many threads it uses or how many clients send requests to it. As such, it is not possible to develop detailed models of the behaviour of these components. However, simpler models are useful for determining important characteristics such as probabilities of successful invocation and expected response

```
// Internal component Technical Analysis, instance 1
module TA1
  t1 : [0..1] init 0;
  [TA1Invoke1] t1=0 -> internalOpRate:(t1'=1);
  [TA1Exec1]   t1=1 -> ta1Rate:(t1'=0);
endmodule
```

Fig. 4.6: Internal TA instance of the parametric CTMC model (FX)

```
// Reward structures
rewards "droppedRequests"
  [NewReq] q=MAX_QUEUE_SIZE : 1;
endrewards

rewards "numOfReqsHandled"
  [NewReq] q<MAX_QUEUE_SIZE : 1;
endrewards

rewards "MWcount"
  [] s1=1 : 1;
endrewards

...

rewards "Ordercount"
  [] s1=6 : 1;
endrewards

rewards "served"
  [ServeReq1] true : 1;
endrewards

rewards "extFails"
  s1=9 : 1;
endrewards

rewards "processingTime"
  s1>0 : 1;
endrewards

rewards "qLen"
  true : q;
endrewards

rewards "servedTA"
  [TA1Invoke1] true : 1;
endrewards

rewards "taTime"
  s1=2 : 1;
  s1=3 : 1;
endrewards
```

Fig. 4.7: Reward structures of the parametric CTMC model (FX)

times. We provide such models for the external services used by the FX system from our motivating example in Section B.1 of Appendix B.

4.2.2 Model Instantiation

Step 2 of the approach instantiates the parametric performance models for combinations of parameter values covering the entire operational profile space. As regularly done, e.g., in the so-called "experiments" performed using the probabilistic model checker PRISM, a suitable discretization of any continuous parameters of the model needs to be used for this purpose. This requires knowing the value ranges that the parameters have when the system is deployed, and deciding the granularity of the discretisation.

The main aim of instantiation in our approach is to select values for the system's parameters that cover its entire operational profile space, enabling the antipatterns'

detection (Section 4.2.4) in regions where the system operates or is expected to operate. There is notable work in the literature on parameter synthesis [59, 93] and on determining optimal parameter settings [151], i.e., which instantiation meets a given objective the best.

For example, the probabilistic model for the FX system from our motivating example has six continuous parameters—$pObjNotMet, pObjSatisfied$, $pObjNotMetHighVar$, $pExpertMode$, $pFAMode$ and $pPerformTransaction$, and we considered the range of possible values $[0.01, 0.99]$ for each of these parameters. A possible discretisation for these intervals (which we used in our experiments) involves assigning to each parameter values starting from the lower bound of their ranges to the upper bound with a step of 0.1 for $pExpertMode$ and 0.1 for $pPerformTransaction$. For $pFAMode$, which is defined as $1 - pExpertMode$, the assignment of parameter values starts from the upper bound of its range to the lower bound with a step of 0.1. The rationale behind choosing 0.1 as a stating value for the above parameters and their increment with a step of 0.1 was to cover the entire operational profile space of the system. The reason for not choosing a smaller step value was due to our experimentation with smaller values reporting no variations in the results. For $pObjSatisfied$ and $pObjNotMet$ we did not explore their whole range, but bounded their values under three cases: $\{(0.21), (0.48), (0.98)\}$ and $\{(0.78), (0.01), (0.01)\}$, respectively. Consequently, for $pObjNotMetHighVar$, which is defined as $1 - pObjSatisfied - pObjNotMet$, the values of the three cases are $\{(0.01), (0.51), (0.01)\}$.

The number of model instances produced in this step can potentially be very high, e.g. for our motivating example we ended up with 30 model instances. This means that the granularity of the discretisation needs to be adjusted so that it is lower when the model has large numbers of parameters, which will impact the accuracy of the analysis from the next step.

4.2.3 Model Analysis

The performance models are then analysed in step 3 to compute the performance indices corresponding to all considered combinations of operational profile parameter values. Existing analysis tools suitable for the adopted type of performance models need to be used in this step–in the case of our DTMC and CTMC models, a probabilistic model checker such as PRISM [109] or Storm [60].

In this analysis step we choose to verify a number of system and component related properties for the CTMC model from Figures 4.3–4.7 that are used in the following sections to identify the occurrence of performance antipatterns across the operational

profile of the system. These properties capture information about the performance and operation of both the overall system and the individual components comprising it. For the FX system from our motivating example, these properties included:

(P_1) $R\{\text{``}droppedRequests\text{''}\} = ?[C <= T] \, / \, T$

Desc.: The dropping rate of requests by the queue in T time-steps.

(P_2) $R\{\text{``}served\text{''}\} = ?[C <= T] \, / \, T$

Desc.: The service rate of requests in T time-steps.

(P_3) $R\{\text{``}processingTime\text{''}\} = ?[C <= T] \, / \, R\{\text{``}served\text{''}\} = ?[C <= T]$

Desc.: The system's average processing time in T time-steps.

(P_4) $(R\{\text{``}qLen\text{''}\} = ?[S]/nthreads + 1) * R\{\text{``}processingTime\text{''}\} = ?[C <= T] \, /$
$R\{\text{``}served\text{''}\} = ?[C <= T]$

Desc.: The system's average response time in T time-steps.

(P_5) $(R\{\text{``}qLen\text{''}\} = ?[S]/nthreads + 1) * R\{\text{``}taTime\text{''}\} = ?[C <= T] \, /$
$R\{\text{``}servedTA\text{''}\} = ?[C <= T]$

Desc.: TA's response time in T time-steps.

(P_6) $R\{\text{``}Cname_count\text{''}\} = ?[C <= T] \, / \, T$

Desc.: Invocation rate of internal/external components (per time unit).

(P_7) $R\{\text{``}Cname_count\text{''}\} = ?[F \, s1 = 5 \, | \, s1 = 9]$

Desc.: Number of internal/external component invocations (per request).

(P_8) $P = ?[\text{``}Path\text{''} \, U \, \text{``}Path_End\text{''}]$

Desc.: The probability of executing a path of the system (i.e, a sequence of states in the model until the state specified as final has been reached).

In these properties, T is the number of time-steps, *Cname_count* refers to any of the internal components or external services of the system (e.g., *MWcount*), *Path* refers to the components of that path (and their corresponding states), e.g., "$MWTAOrNoPath$" $= (s1 = 0 \, | \, s1 = 1 \, | \, s1 = 2 \, | \, s1 = 3 \, | \, s1 = 6 \, | \, s1 = 7)$, and *Path_End* refers to the last state of that path; e.g., in the previous example $s1 = 7$. Example graphs for some of these properties, generated while varying the operational profile parameters of the system, can be found in Section B.3 of Appendix A.

4.2.4 Antipattern Profile Generation

Step 4 of the approach is using the performance indices and a portfolio of antipattern detection rules to identify the performance antipatterns that occur for different combinations of parameter values. This step produces a series of maps that show the distribution of such antipatterns across the operational profile space, thus highlighting problematic (from a performance viewpoint) areas.

4.2.4.1 Antipattern Detection Rules

The concept of Performance Antipattern has been introduced several years ago [149] to define bad design practices that can induce performance problems in software systems. This concept has been later formalized in First-Order Logics [51] and then employed, in the context of Software Performance Engineering processes, for the purpose of automating the detection and solution of performance problems [138]. More information regarding the concept of performance antipatterns along with definitions for the ones used in this chapter can be found in Section 2.3.

Inspired by the formalization provided in [51], we have here bounded the detection rules of three performance antipatterns to the modeling and analysis context of this approach. This binding is indeed required for any context, due to specificities and possible limitations of the notations adopted. In our case, Markov models of service-based software systems, on one hand, offer the advantage of easy analysis of component and system properties of interest and, on the other hand, suffer of lack of separation between software and hardware parameters. The latter are in fact implicitly taken into account in execution rates of operations.

Hereafter we report the formalization of the performance antipattern detection rules that we have used in this approach, while their parameters are defined in Table 4.1, where we also specify whether each parameter is available for external services (‘EXT’), for internal components (‘INT’), or for both (‘EXT/INT’).

- **BLOB**

  `General description`

  It occurs when a component performs most of the work of an application, thus resulting in excessive components’ interactions that can degrade performance.

  `Internal components`

  $(InvReq > AvgInvReq) \land (Util > UtilThresh) \land (Util > AvgUtil)$

Table 4.1: Detection rule parameters.

Variable	Scope	Description
InvReq	EXT/INT	Number of invocations per request
AvgInvReq	EXT/INT	Average number of invocations per request
InvTime	EXT/INT	Number of invocations per time unit
AvgInvTime	EXT/INT	Average number of invocations per time unit
ServRate	INT	Service rate
Util	INT	Utilization
AvgUtil	INT	Average utilization
UtilThresh	INT	Fixed utilization threshold
RespTime	EXT	Response time
AvgRespTime	EXT	Average response time
PathProb	EXT/INT	Probability of path execution
AvgPathProb	EXT/INT	Average probability of path execution
PathProbThresh	EXT/INT	Fixed threshold for probability of path execution

`External components`

$InvReq > AvgInvReq$

- **CONCURRENT PROCESSING SYSTEMS (CPS)**

`General description`

It occurs either when too many resources are dedicated to a component (MAX) or when a component does not make use of available resources (MIN).

`Internal components`

MAX - $(Util > UtilThresh) \wedge (Util > AvgUtil)$

MIN - $(Util < UtilThresh) \wedge (Util < AvgUtil)$

`External components`

MAX - $PAR\ pattern \wedge (RespTime > AvgRespTime)$

MIN - $PAR\ pattern \wedge (RespTime < AvgRespTime)$

- **PIPE AND FILTER (P&F)**

`General description`

It occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput.

`Internal and External components`

$$(InvTime > AvgInvTime) \wedge (PathProb > PathProbThresh) \wedge$$
$$\wedge (PathProb > AvgPathProb)$$

We remark that, in our context, the rules for detecting a specific antipattern on internal components may differ from the ones defined for external services. This is because the parameters available for external services are obviously more limited than those of the internally developed components. For example, the whole response time (i.e., service plus waiting time) of an external service is usually negotiated in a service-level agreement, but it is difficult to isolate the net service time contribution to it, due to lack of control on the execution platform and the amount of resources dedicated to the service by the provider. Both indices can instead be estimated for internal components. As a consequence, wherever the service time (or any derived index like utilization) appears in a detection rule, the corresponding predicate has to be skipped/modified for external services. For this reason, in our case BLOB and CPS antipatterns have different rules when applied to internal components or external services because, as reported in Table 4.1, utilization cannot be estimated for the latter ones. In the BLOB case, the predicates including utilization for internal components are simply skipped in the external service formulation, because no other predicate would make sense there. Instead, in the CPS case, the predicates on utilization have been replaced with similar ones on response time for external services, because the CPS definition is compliant with this modification.

We highlight that all predicates include parameters that evidently change across different areas of the system operational profile (e.g., *InvReq*, *Util*), hence we expect that the occurrences of the corresponding antipatterns vary. The only exceptions are the CPS rules for external services, because their parameters and thresholds do not depend on the operational profile. Such rules refer to the response time that, for these components, is based on service level agreement, and thus it cannot vary with the operational profile. This will evidently reflect on our experimental results, where CPS on external services will appear either everywhere or nowhere in the operational profile space.

As previously mentioned, in our approach we focused on the detection of three performance antipatterns. However, we expect that the approach can be used with any combination or all of the available performance antipatterns listed in Section 2.3. There has been existing work in the literature aiming at addressing performance problems in software systems by deploying modelling techniques, specifically queuing network models [111], in order to provide evidence of how antipatterns may affect the overall

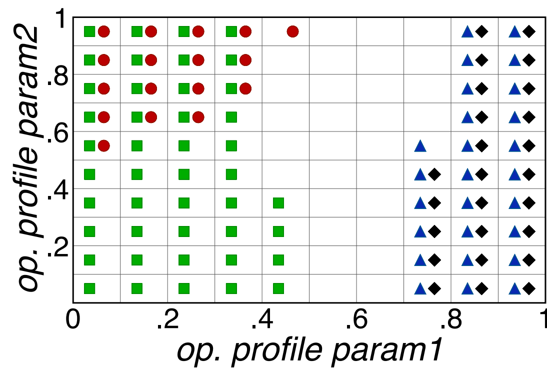


Fig. 4.8: Example of *antipattern profile*

system performance (e.g., [48, 54, 130]). Queuing networks can be expressed as Markov chains (e.g., [101, 153]), giving us the confidence to pursue an extension of our current approach in the future.

The reason for not expanding the list of detection rules has to do primarily with the system we choose to demonstrate the applicability of our approach. The appearance of antipatterns is related to the type of the software system and to the various components comprising it. Thus, it is highly unlikely that all existing antipatterns are applicable to a given software system. Later in the evaluation of the approach (Section 4.4), we mention two additional antipatterns that were included in our initial experimentation, but our analysis results indicated that they do not affect the performance of the FX system.

4.2.4.2 Synthesis of Antipattern Profiles

The more software applications are being used worldwide from different types of users, the more difficult is to estimate a representative average behaviour of users that induces a specific operational profile. In fact, not only can users have different operational profiles depending on their locations [39], but even in the same area the users behaviour can (sometime radically) change over time [98].

Nevertheless, applications should show acceptable performance across different operational profiles. A motivation for our work is that different operational profiles can induce various performance problems, for example because a higher execution frequency of a path can overload components involved in that path. Hence, the idea is that, in order to identify the most appropriate refactoring actions to apply for overcoming performance problems, these problems must be identified across different operational profiles.

In this work, we introduce the concept of *Performance Antipattern Profile*, which is a representation of performance antipattern occurrences while varying operational profile parameters. As discussed above, different antipattern occurrences are expected to appear in different areas of an operational profile. For instance, Figure 4.8 shows two operational profile parameters which vary (from 0 to 1) on the axes, and different coloured shapes in the graph indicate the occurrences of different antipatterns. Only with this information in hand, the performance experts can suggest appropriate refactoring actions when the system falls within a certain operational profile area, or even (in a proactive way) when the system is expected to enter a specific operational profile area.

4.2.5 Refactoring

Finally, step 5 assesses whether refactoring actions are required, because performance antipatterns occur in regions of the operational profile space where the deployed system is expected to operate. When refactoring is required, suitable refactoring actions (selected from a repository of such actions) are used to update the system design. Updated system designs are then further evaluated through re-executing the five steps of the approach, until a design with suitable performance antipattern profiles is obtained.

The notational aspects outlined in the previous section for antipattern detection obviously reflect in the portfolio of refactoring actions aimed at removing performance antipatterns. In general, a refactoring action modifies some available *architectural knob* (e.g., the number of messages exchanged between two components, the list of operations provided by a component) to remove a source of the antipattern causes. The type and number of knobs depend on the adopted notation, so the portfolio of refactoring actions does the same.

Our notation distinguished between internal components and external services. The two types of system elements are characterized by a few common parameters and by parameters specific to each type (see Table 4.1). Therefore, our portfolio of refactoring actions is partitioned in two sets, as detailed below.

Actions for internal components

- `Change service rate` - The modification of a component service rate can be induced by several actions on the system, which could act on the hardware platform or on the software architecture, such as: (i) redeploy the component to a platform node with different hardware characteristics, (ii) replace some devices of the platform node where the component is currently allocated, (iii) redesign the

software component so that its resource requests change, (iv) split a component into two (or more) components and re-deploy them.

- `Change number of threads` - This action is always possible where the control on the number of threads is on the designer's hands, and indeed for internal components this is guaranteed.

Actions for external services

- `Change pattern` - We have considered three combination patterns for external services, that are: SEQ, PAR, and PROB (see Section 4.1.2). They are used to combine (a subset of) the available instances of a certain external service. This action requires to modify the combination pattern, by keeping unchanged the set of combined services.
- `Change the pattern parameters` - Some patterns are regulated by parameters, in particular: PROB has a probability of each instance invocation, and SEQ has a failure probability for each instance. A change in the PROB probabilities is always feasible, because they are under full control of the designer. Instead, a change in the failure probabilities within a SEQ pattern implies that the designers are able to carry out deeper modifications in the involved instances that can induce different reliability, and this is not often the case.
- `Change combination of service instances` - This action requires to replace some (or all) of service instances that are combined to provide a certain operation, by keeping unchanged the combination pattern.

Of course, the above actions can be combined together to study their joint effects on the performance improvement.

In general, identifying refactorings that are relevant to specific antipatterns requires domain knowledge. However, suitable actions have been proposed in the literature to guide software engineers into making the right decisions (see Section 2.3). Additionally, there is published research in the area of performance antipatterns that focuses on formalising the refactoring embedded into performance antipattern definitions, pointing to potential sources of performance problems and suggesting refactoring actions in order to remove these problems [6]. To further support the selection of the most optimal refactoring action from a performance perspective, the work in [159] employs visualisation techniques to assess the impact of each refactoring.

Understanding what are the most suitable refactoring actions for a system's architecture is not a trivial task, and providing further guidance for the selection process is definitely an interesting line of future work for our approach.

4.3 Implementation

To automate the synthesis of antipattern profiles for the system in our motivating example, we implemented an application-specific tool that (a) enables the generation of both DTMC and CTMC models for the external services and the entire system, respectively, (b) verifies properties of interest via probabilistic model checking, and (c) generates maps that show the appearance of performance antipatterns across the operational profile space of the system. The implemented tool, the models and the full experimental results are available at: https://github.com/Fase20/automated-antipattern-detection.

To achieve the full automation of the approach, including the representation of the architecture of an arbitrary application as probabilistic model(s), the identification and application of suitable refactoring actions when antipatterns do appear, non-trivial work is required. Nevertheless, existing research has already achieved some of these objectives and can potentially be adapted into our approach. Specifically, the work in [161] and its extension in [162] makes use of the Palladio Component Model (PCM) modelling language [16] to capture information about component-based software architectures. The PCM models are then used to synthesise performance models whose analysis leads to automatically detect and solve performance antipatterns. There is also information provided with regard to which antipatterns can be automatically detected by using this approach, and whether they can also be automatically solved.

Furthermore, [55] and [99] employ Unified Modeling Language (UML) diagrams [114] to model the architecture of software systems, annotated with nonfunctional characteristics. The former introduces a framework to jointly model and analyse the security and performance attributes of software architectures. The latter presents a performance modeling framework that enables the generation of CTMC models from a system specification description captured by UML diagrams. The CTMCs are then analysed to validate the performance requirements set at an early stage of the system's development life cycle.

The above cited research proves that full automation of our approach can be achieved, and there are a few possible directions that one can consider adopting into this work in the future.

Table 4.2: System parameters.

Parameter	Values
ExtReqs-rate	$10s^{-1}$
QueueSize	10
TA-rate	$3s^{-1}$
Alarm-rate	$40s^{-1}$
Notif-rate	$55s^{-1}$
MW-rate	$19.92s^{-1}$
FA-rate	$24.99s^{-1}$
Order-rate	$19.09s^{-1}$
TA-threads	1

4.4 Evaluation

In this section, we first introduce the research questions that we intend to address (see Section 4.4.1). Thereafter, we describe the experimental scenarios (see Section 4.4.2) and discuss the obtained results (see Section 4.4.3). We finally report the threats to validity in Section 4.4.4.

4.4.1 Research Questions

The detection and solution of performance antipatterns largely depends on the operational profile, which is determined by the end-users behaviour, thus it can only be known after the system deployment. Naturally, some antipatterns are more affected than others by the operational profile that can have a considerable influence on the software system and, consequently, on its performance characteristics. Through our experimentation, we aim at answering the following two research questions:

- RQ_1: Does our approach provide insights on the performance antipattern profile of a specific design?
- RQ_2: Does our approach support performance-driven refactoring decisions on the basis of the performance antipattern profile?

In order to answer these questions, we apply our approach to the running example introduced in Section 4.1.

4.4.2 Experimental Scenarios

Table 4.2 reports the system parameters of the default configuration we have used for our experiments. It is structured in three different groups. First, system settings, i.e., *ExtReqs-rate* (rate of external requests incoming to the system), and *QueueSize* (maximum number of queueing requests). These values are both set to 10. Second, the rate of internal components and external services, e.g., $TA\text{-}rate = 3$ is the execution rate of the Technical Analysis (TA) internal component. For external services, this rate corresponds to the inverse of the response time (as explained in Section 4.2.4.1), and it was obtained through the analysis of discrete-time Markov chain (DTMC) models of the service combinations (i.e., SEQ, PAR or PROB) used for the external operations of the system. The model checker Storm was used to perform this analysis. The choice of the rates values was based on reflecting the operation of each internal component and external service. For instance, the Alarm internal component, when triggered, will immediately serve its purpose without a significant computational cost. Thus, it has a higher rate than the FA external service, which needs to perform an analysis before reporting back the results. Third, TA (as internal component) has a number of threads that is initially set to 1, but we provide a refactoring action that can change such number to modify the parallelism degree for such component.

The operational profile space of our running example (see Figure 4.1) is fully defined by the following branching point probabilities: (i) `pExpertMode` (p_{EM}), i.e., the probability of executing the workflow in expert mode; (ii) `pPerformTransaction` (p_{PT}), i.e., the probability of successfully performing a transaction; (iii) `pObjecti-vesSatisfied` (p_{OS}) and `pObjectivesNotMet` (p_{ON}), i.e., the probabilities of satisfying or not the objectives, respectively. As a consequence, $1-(p_{OS}+p_{ON})$ is the resulting probability of an error occurring.

The experimental scenarios that we analyze in the next section include the variations of p_{EM} and p_{PT} within their full range $[0,1]$ with a 0.1 step. Additionally, we decided to bind (p_{OS}, p_{ON}) to three scenarios, namely: $\{$ (0.21, 0.78), (0.48, 0.01), (0.98, 0.01) $\}$, which in the following we call $scenario_A$, $scenario_B$, and $scenario_C$, respectively.

We have considered the following design changes for refactoring purposes:

(R_1) the service rate of the TA internal component can be modified from 3 to 6 jobs per second (i.e., it becomes faster when performing computations) when TA is detected as an instance of a BLOB performance antipattern;

(R_2) a further thread of the TA component can be added to split the incoming load and manage users' requests, again as a solution of a BLOB performance antipattern on TA;

(R_3) change pattern (from SEQ to PAR) and service rate (from 50.21 to 500) of the MW external service, when MW has been detected as part of a Pipe and Filter antipattern;

(R_4) change service rate (from 40.02 to 400) of the FA external service while keeping the same pattern (i.e., PAR), and this is suggested as a solution of a Pipe and Filter antipattern that involves FA.

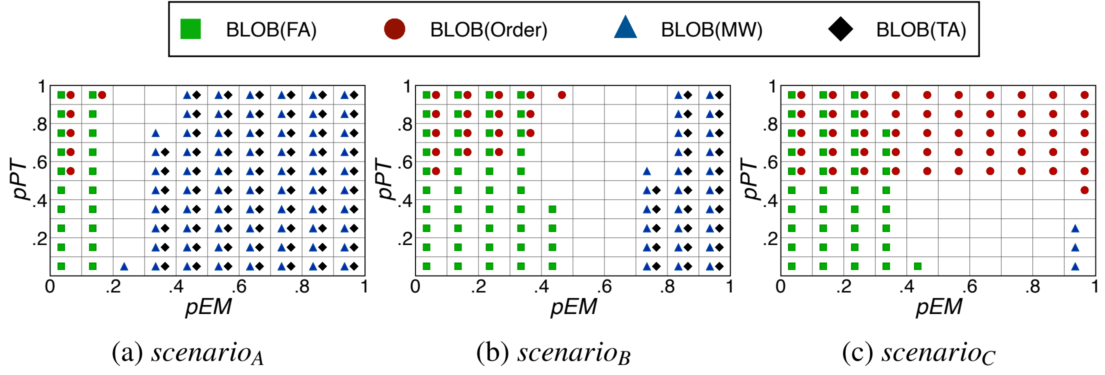
The results presented in the next section were obtained using the tool we developed to implement the analysis and refactoring process from Figure 4.2. This tool generates antipattern profiles using the antipattern detection rules from Section 4.2 and performance indices computed through the probabilistic model checking of a continuous-time Markov chain (CTMC) model of the entire FX system from Figure 4.1. The model checker Storm is automatically invoked by the tool for this purpose. The tool and the parametric CTMC models we used are available in our project's GitHub repository.

4.4.3 Experimental Results

In order to answer RQ_1, we have investigated the occurrence of performance antipatterns across different operational profiles, so as to obtain performance antipattern profiles. Figures 4.9, 4.10, and 4.11 report the BLOB, CPS, and P&F detected antipatterns, respectively, across the operational profile space. Each figure shows the three considered scenarios for p_{OS} and p_{ON} and, for each scenario, p_{EM} varies from 0 to 1 (with a step size of 0.1) on the x-axis, while p_{PT} varies in the same range on the y-axis. Antipatterns occurring in each operational profile point are denoted by specific symbols.

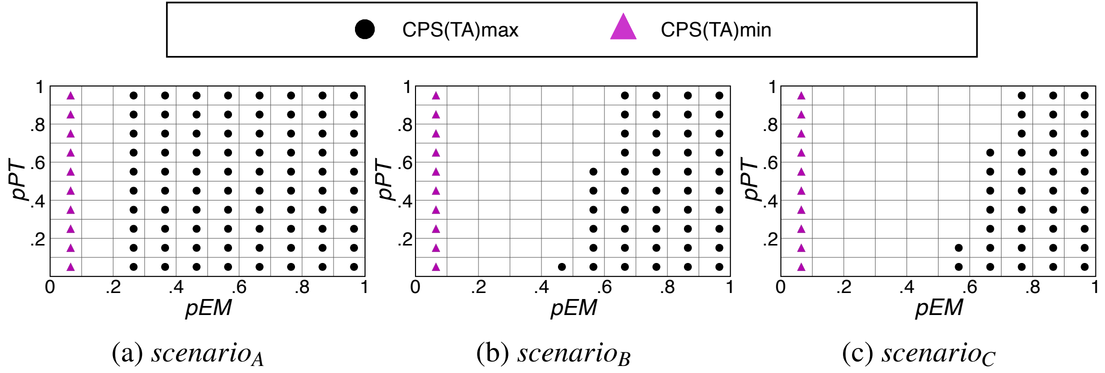
We have here considered full ranges of the operational profile parameters, even though, in each instant of its runtime, the system will fall in a single point of the profile. Therefore, suitable refactoring actions depend on the area where the running system profile falls in the considered time. In particular, if it runs in an area where antipatterns do not occur, then no refactoring action is suggested.

In Figure 4.9(a) we can notice that in $scenario_A$ (i.e., $p_{OS} = 0.21$ and $p_{ON} = 0.78$) four different components are detected as BLOB antipatterns, specifically: (i) BLOB(FA) occurs for low values of p_{EM} only (i.e., up to 0.2); as opposite, (ii) BLOB(TA) occurs for larger values of p_{EM}; (iii) BLOB(MW) shows a very simi-



(a) $scenario_A$ (b) $scenario_B$ (c) $scenario_C$

Fig. 4.9: BLOB antipattern instances while varying operational profiles.



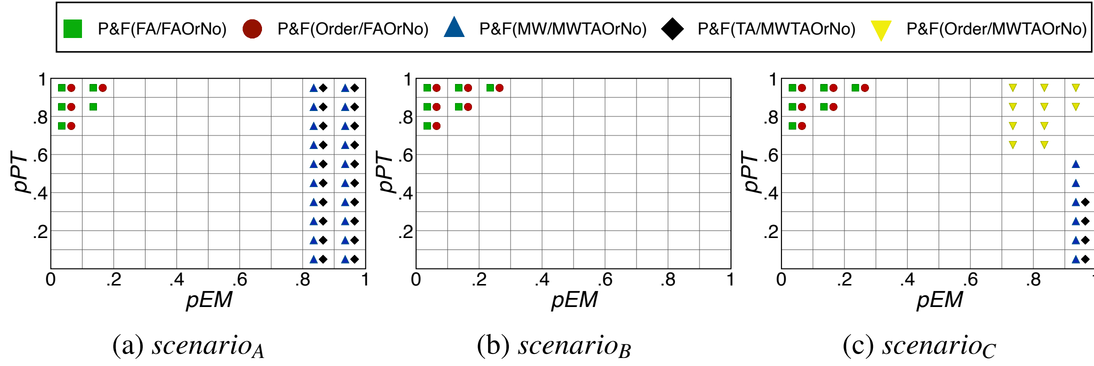
(a) $scenario_A$ (b) $scenario_B$ (c) $scenario_C$

Fig. 4.10: CPS antipattern instances while varying operational profiles.

lar behaviour with respect to BLOB(TA) except in two corner cases where it occurs alone; (iv) BLOB(Order) occurs for low values of p_{EM} and high values of p_{PT} only.

Figure 4.9(b) interestingly shows that in $scenario_B$ (i.e., $p_{OS} = 0.48$, and $p_{ON} = 0.01$), BLOB(TA) and BLOB(MW) occur in a smaller portion of the operational profile space, i.e., the right-most side (starting when p_{EM}= 0.7). Also the other antipatterns are subject to the probability changes, in fact both BLOB(FA) and BLOB(Order) occur in a larger portion of the space, i.e., the left-most side (up to p_{EM}=0.5). This is because $scenario_B$ moves a consistent part of the workload far from the MW-TA loop, with respect to $scenario_A$.

Figure 4.9(c) illustrates the case of $scenario_C$ (i.e., $p_{OS} = 0.98$, and $p_{ON} = 0.01$), where further differences appear. In particular, BLOB(TA) antipattern does not occur anymore since the higher value of p_{OS} induces less computation in TA. BLOB(MW) is confined to three cases of large p_{EM} values and low p_{PT} values. This is because the major load is going here to FA and Order that in fact more widely are detected as BLOB antipatterns.



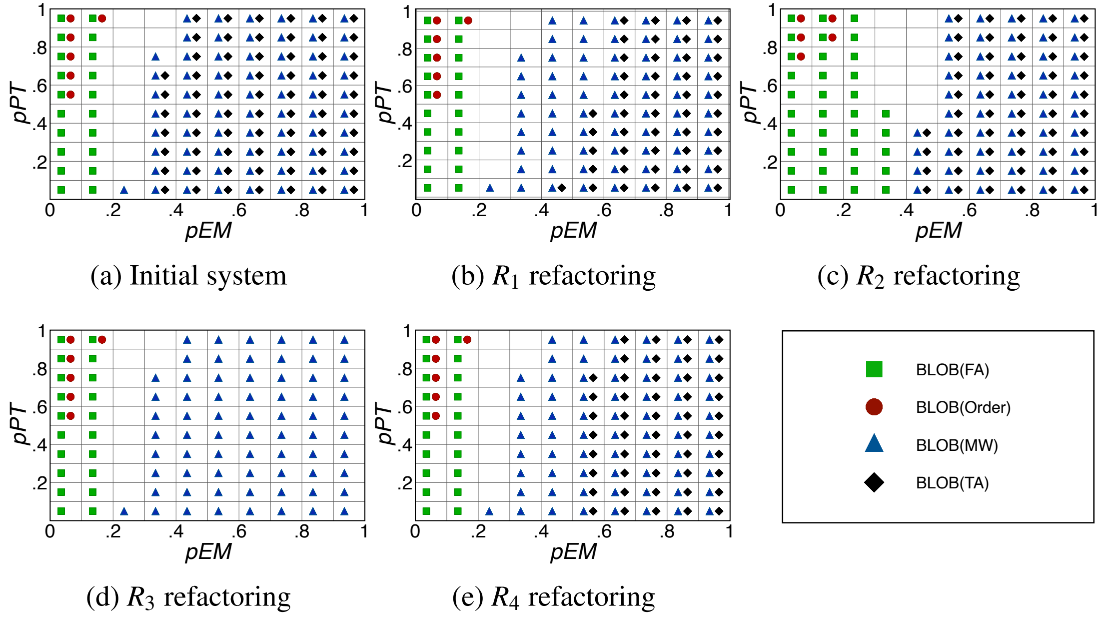
(a) $scenario_A$ (b) $scenario_B$ (c) $scenario_C$

Fig. 4.11: P&F antipattern instances while varying operational profiles.

Figure 4.10 depicts the CPS antipattern profile that, as compared to the BLOB one, does not considerably vary across different scenarios. For readability reasons, CPS(FA)min is not reported in this figure, although it occurs across the whole operational space for all the three scenarios. We recall that this is due to the CPS detection rule that takes into account the response time for external services, which does not change with users' behaviour since it is a fixed value outcoming from service-level agreements. CPS(TA)min is not affected at all by the scenario variations, as it always occurs in the same operational profile area. Instead, the CPS(TA)max instances progressively decrease when increasing p_{OS}. A p_{OS} growth, in fact, relieves the MW-TA loop, thus inducing less unbalancing in its components.

Figure 4.11 shows the P&F antipattern profile, where the antipattern instances obviously refer to execution paths instead of single components/services. Hence, different symbols represents different paths where one of the components/services is the slowest filter. For example, MW/MWTAOrNo means that MW is the slowest filter of the MW-TA-Order-Notification path. Interesting variations of this antipattern profile appear across scenarios, again driven by variations in the operational profile parameter values.

Summary for RQ_1: The above experimentation shows the ability of our approach in identifying performance problems in a given software system through the appearance of performance antipatterns. These antipatterns are reported using the detection rules from Section 4.2.4.1. The violation of a rule indicates the presence of a performance problem within the system as per antipattern's description. We also observe that the antipatterns' appearance is dependent on the system's operational profile. Figures 4.9–4.11 act as evidence for the previous statement as we observe antipattern variations for different values of operational profile parameters. Our approach provides insights on the performance antipattern profile of a specific design.



(a) Initial system (b) R_1 refactoring (c) R_2 refactoring

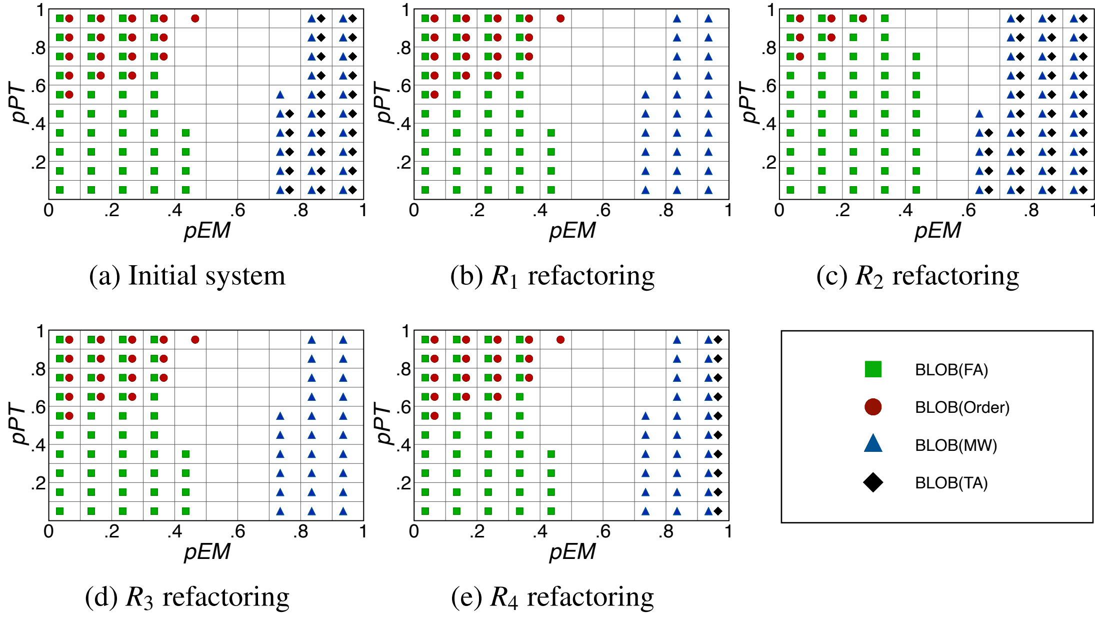
(d) R_3 refactoring (e) R_4 refactoring

Fig. 4.12: BLOB antipattern instances across different refactorings - $scenario_A$.

> In fact, we are able to identify considerable variations in the detected antipattern instances while varying the operational profile parameters.

In order to answer RQ_2, we have investigated the occurrence of performance antipatterns after applying refactoring actions that we have defined in Section 4.4.2, across the operational profile space. The most interesting cases are discussed hereafter, and specifically: (i) Figures 4.12 and 4.13 report the BLOB refactoring effects on $scenario_A$ and $scenario_B$, respectively; (ii) Figure 4.14 illustrates refactorings for the CPS antipattern in $scenario_A$; (iii) Figure 4.15 shows the P&F refactoring effect on $scenario_C$.

In Figure 4.12, we can notice the following effects of refactorings actions. Upon (R_1) application, as expected, less BLOB(TA) instances appear because this refactoring consists of doubling the TA computation speed, while all other instances remains unvaried. (R_2) introduces a further TA thread and, in this case, this induces less BLOB (TA) because more quickly requests are processed by these two threads, and realistically FA becomes the overloaded one thus inducing more BLOB(FA) instances to appear. (R_3) modifies the rate of MW and makes it much slower, thus inducing the side effect of providing much less load to TA; in fact all the BLOB(TA) instances disappear, and all the other instances remain unvaried. (R_4) decreases the rate of FA and, similarly to above, it has the effect of providing less load to TA, in fact the number of BLOB(TA) instances decreases.



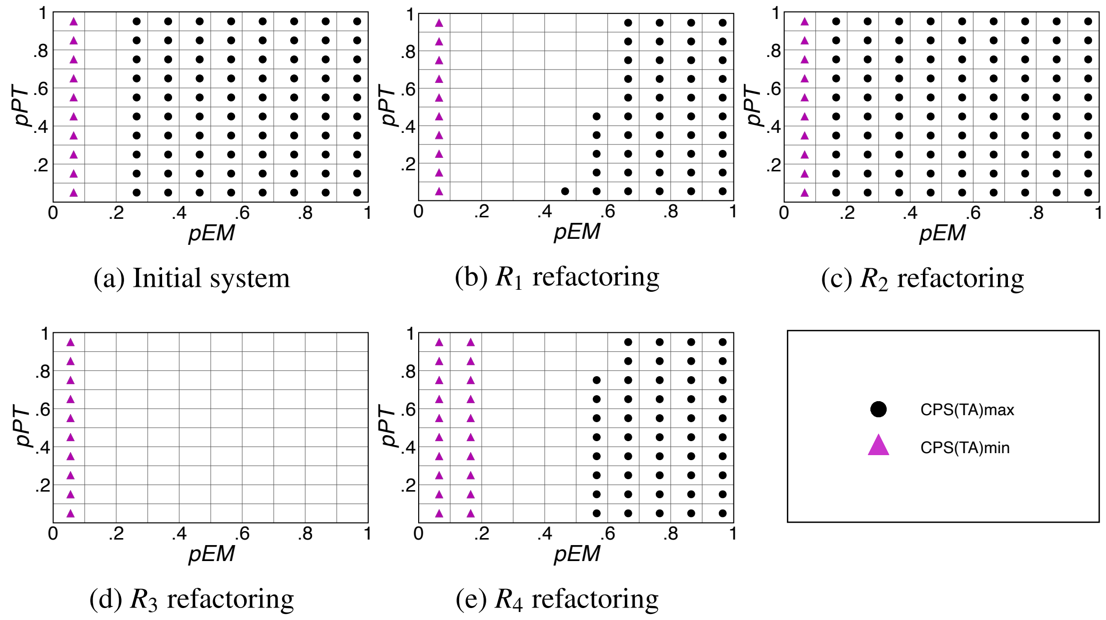
(a) Initial system (b) R_1 refactoring (c) R_2 refactoring

(d) R_3 refactoring (e) R_4 refactoring

Fig. 4.13: BLOB antipattern instances across different refactorings - $scenario_B$.

Figure 4.13 illustrates the effect of BLOB refactorings on $scenario_B$. (R_1) refactoring consists of making the TA component two times faster, hence the BLOB(TA) instance completely disappears from the operational space, while all the other antipatterns are not affected. (R_2), introduces a further TA thread, but in this case it occurs in a quite less stressed context with respect to $scenario_A$. This aspect, together with the fact that two threads allow to drop less requests, given that the queue length remains unvaried, in practice does not relieve TA itself. This is the reason for BLOB(TA) to not disappear. The decrease of BLOB(Order) instances is very likely due to the fact that, if performance indices change for some components/services, then their calculated average value change as well, hence inequalities in detection rules can change their results due to changes in the right-hand-side targets. (R_3), similarly to Figure 4.12, modifies the MW rate and makes it much slower, thus having the effect of providing much less load to TA, in fact all BLOB(TA) instances disappear. Also (R_4) behaves similarly to Figure 4.12.

Figure 4.14 depicts $scenario_A$ (i.e., the $p_{OS} = 0.21$ and $p_{ON} = 0.78$ case) when considering CPS antipattern instances. We recall that the detection rule for CPS on external services operates on response time values that do not change with the operational profile. This leads that CPS(FA)min occurs in the whole operational space (not only for the initial system, but also after R_1, R_2, and R_3 refactorings). Instead, for R_4 refactoring, we found CPS(FA)max always occurring, and this is due to nature of this refactoring that modifies the FA rate. For R_3 refactoring, besides CPS(FA)min, we



(a) Initial system (b) R_1 refactoring (c) R_2 refactoring

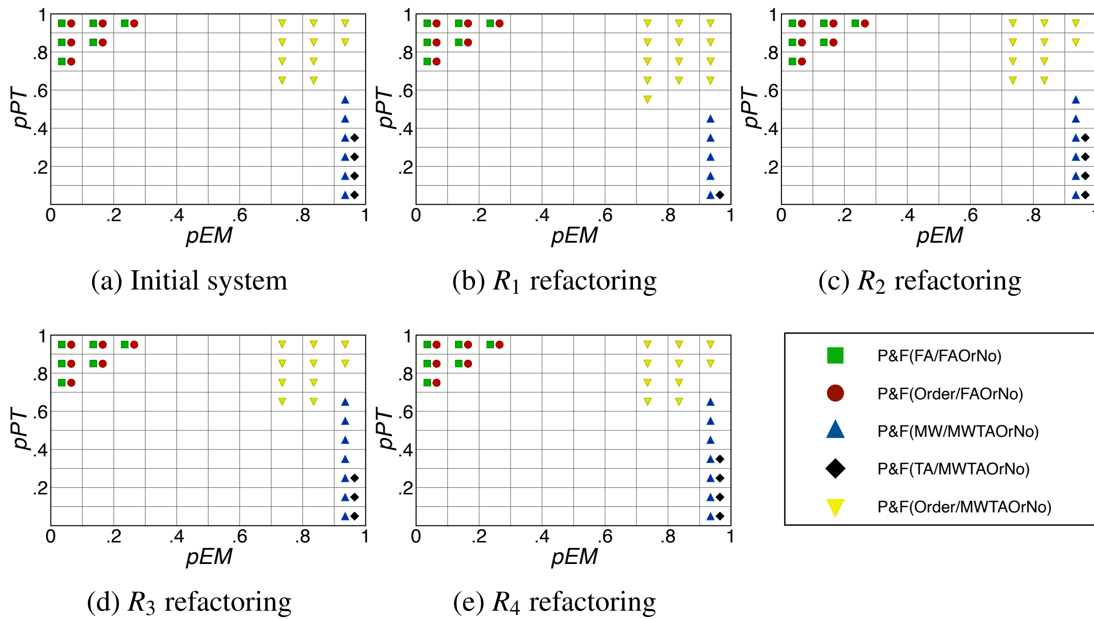
(d) R_3 refactoring (e) R_4 refactoring

Fig. 4.14: CPS antipattern instances across different refactorings - $scenario_A$.

also found CPS(MW)max always occurring, and this is again due to the fact that R_3 modifies the MW rate.

In addition to this, we can make the following specific considerations. (R_1), makes the TA component two times faster, hence less CPS(TA)max instances appear, as expected. (R_2) introduces a further TA thread but it is not beneficial for the system, in fact the number of CPS(TA)max instances increase in the operational profile space. This effect is again very likely due to the fact that, with two threads, less requests are dropped than in the one thread case. Hence the work on TA in practice increases. This apparent anomaly would be mitigated whether, in the analysis, the number of dropped requests would be considered. (R_3), decreases the MW rate, so it has the effect of providing less load to TA; in fact CPS(TA)max instances disappear, and (as mentioned above) a CPS(MW)max instance appears in the whole operational profile space. (R_4) decreases the FA rate, thus having the effect of increasing the number of CPS(TA)min instances and decreasing the CPS(TA)max ones.

Figure 4.15 illustrates $scenario_C$ (i.e., the $p_{OS} = 0.98$ and $p_{ON} = 0.01$ case) when considering P&F antipattern instances. Quite small variations can be observed here, as compared to other antipatterns and scenarios, always limitedly to single points of the operational profile space. Some specific comments follow. (R_1) induces less P&F instances where TA is the slowest filter and, on the same path, introduces more instances where Order is the slowest filter. This is an expected behaviour due to the refactoring action that makes TA faster. (R_2) has no effect at all. (R_3) modifies the rate of MW



(a) Initial system (b) R_1 refactoring (c) R_2 refactoring

(d) R_3 refactoring (e) R_4 refactoring

Fig. 4.15: P&F antipattern instances across different refactorings - $scenario_C$.

component and makes it much slower, thus inducing less load to TA. The effect on the P&F antipattern is minimal and coherent, because one more P&F(MW) instance and one less P&F(TA) instance occur in the same path. (R_4) only introduces one more P&F(MW) on the same path as above, and this could be a side effect of changing the average values of performance indices.

Summary for RQ_2: The above experimentation with the applied refactoring actions demonstrates the capability of our approach to support performance-driven refactoring decisions based on antipattern profiles. Refactorings yield different effects on different regions of the system's operational profile space and that is evident in Figures 4.12–4.15. As a result, performance antipattern profiles can inform the system's engineers about the impact of an applied refactoring and assist them with making the right decisions towards ensuring the system's performance objectives.

4.4.4 Threats to Validity

Construct validity. To mitigate the threats that may arise from the construction of the parametric DTMC and CTMC models, we adapted a case study that has already been applied in other research work [34], and modelled the corresponding system as a parametric CTMC model following the diagram of Figure 4.1. For the modelling of the parametric DTMCs representing external services, we took into consideration that

multiple services can be used as equivalent alternatives, thus, we used three options for combining these services as detailed in Section 4.1.2.

Internal validity. In order to spot internal errors in our implementation for automatically detecting multiple performance antipatterns, we have thoroughly tested it. First, we ensured that the results obtained from the verification of the formalised properties, located in Section 4.2.3, represent the expected values for each combination of parameter values. Secondly, we verified that the detected performance antipatterns follow the given rules defined in their specification (Section 4.2.4.1), along with the expected performance indicators. Finally, we assessed the above in scenarios involving the detection of three performance antipatterns, with different detection rules for antipattern instances related to internal components and external services of the system. Note that the detection and solution of performance antipatterns relies on our previous experience in this domain [51], but in the future we are interested to involve external users that will be enabled to add their own rules for detection and refactoring.

External validity. We are aware that one case study is not enough to thoroughly validate the effectiveness of our approach. Nevertheless, several experiments have been performed beside the proposed experimental scenarios, in order to inspect the large number of variabilities in the operational profile space that may affect performance characteristics in unexpected ways. These experiments were performed to further assess the correctness of our approach and produced similar results with the ones displayed in Section 4.4.3. As future work, we would like to better investigate the effectiveness of our approach by applying it to further case studies (including industrial applications).

4.5 Related Work

In the literature, the operational profile of a software system has been recognized as a very relevant factor in many areas of software engineering, including software reliability [126] and testing [142]. In the context of performance analysis of software systems, there are many techniques developed to act at: (i) design-time, i.e., providing model-based predictions [18, 32, 156]; and (ii) run-time, i.e., actual measurements derived from system monitoring [29, 33, 170]. Software refactoring, instead, is a more recent research direction, and many issues arise when modifying different system abstractions [3, 125, 15]. This approach contributes in demonstrating that both performance analysis and refactoring are affected by operational profiles, and in the following we review the related work aimed at pursuing this research direction.

In [94], a method for uncertainty analysis of the operational profile is presented, and the perturbation theory is used to evaluate how the execution rates of software components are affected by changes in the operational profile. Our approach also considers execution rates, but it is intended to support designers in the task of identifying performance-critical scenarios (i.e., when antipatterns occur and their evolution when refactoring actions are applied). In [167], performance antipatterns are used to isolate the problems' root causes, and facilitating their solutions; the TPC-W benchmark showed a relevant increase in the maximum throughput, proving the usefulness of performance antipatterns. However, the choice of representative usage profiles is recognized by the authors as a limitation of the approach, since no directives are given for this scope. Our approach, instead, is intentionally focused on exploiting the performance antipatterns while considering the operational profile space as a first-class citizen of the conducted analysis.

The static technique proposed in [122] detects and fixes performance problems (i.e., break out of the loop when a given condition becomes true). It has been applied to real-world Java and C/C++ applications, producing very promising results since a large number of new performance bugs were discovered. Like [167], this approach neglects the operational profile that instead may trigger the presence of further performance problems. In contrast, our goal is to shed light on the importance of the operational profile space, and our experimentation demonstrates that performance problems and solutions indeed vary across this space.

In [77], performance anomalies in testing data are detected through a new metric, namely the transaction profile (TP), that is inferred from the testing data along with a queuing network model of the testing system. The key intuition is that TP is independent from the workload, and sensitive to variations caused by software updates only. Our approach also investigates which refactorings are more responsible for performance issues, along with the characteristics of the operational profile. In fact, refactorings produce regions of the operational profile space that are differently affected, and these differences can be used by the designers to understand the suitability of a specific design.

The work most related to our approach is [133], where sequences of code refactorings (for Java-like programs) are driven by the avoidance of antipatterns (i.e., the BLOB only) and aimed at improving the system security. These refactorings consider the attack surface (i.e., how users/attackers access to software functionalities) as an additional optimization objective. Our approach shares the intuition that antipattern-based refactorings are beneficial for software quality (i.e., performance in our case)

and that the operational profile needs to be part of the evaluation. Unlike [133], we target software design abstractions, and we provide a global view of the antipatterns encountered by software systems across their entire operational profile space.

A systematic literature review on software architecture optimization methods is provided in [1], but users' operational profiles are neglected. This further motivates our work as promoter of a research line that should foster more attention on the role of users and their effects on the available software resources.

Summarizing, to the best of our knowledge, there is no approach that focuses on how the operational profile affects the performance analysis and refactoring of software systems, and the idea of adopting performance antipatterns for this scope seems to be promising according to our experimentation.

4.6 Summary

We presented a novel approach that considers the operational profile space of a system under development as a first class citizen in performance-driven analysis and refactoring of software systems. Performance antipatterns profiles have been used to support designers in the nontrivial task of identifying problematic (from a performance perspective) areas of the operational profile space, and refactoring actions are applied to improve the system performance in such areas. Experimental results confirm the usefulness of the approach, and show how it can be used to evaluate the suitability of a specific design in different regions of the operational profile space.

Chapter 5

Software Performance Engineering with Code-level Probabilistic Analysis and Performance Antipatterns

As explained in earlier chapters of this thesis, much of the recent research on software performance has focused on the architecture of software systems [1, 6, 74]. The code analysis technique introduced in Chapter 3 complements these architecture-level analysis methods. However, both architecture-level and code-level software performance engineering are important, and need to be used in conjunction for best results.

This chapter introduces a methodology that integrates these two types of analysis in a novel way, by combining the performance analysis and refactoring techniques from Chapters 3 and 4, based on probabilistic modelling [95] and performance antipatterns [144], respectively. The combination of the two techniques enables the use of the integrated methodology at both code-level and system architecture-level. PROPER analyzes code performance, yielding verification results that then support the modelling of the system at architectural level, and the antipattern-based analysis method performs system-level performance analysis that guides refactoring using synthesised performance antipattern profiles.

The remainder of this chapter is organised as follows. Section 5.1 presents a motivating example that is used to illustrate the applicability of the integrated methodology. Section 5.2 presents the aforementioned hybrid methodology for software performance engineering. The first step of the methodology (*A* in Figure 5.2) involves the identification of system components of interest (Section 5.2.1). The second step (*B* in Figure 5.2) employs PROPER to verify performance properties of these components at code-level

(Section 5.2.2). The third step (*C* in Figure 5.2) provides the results obtained during the previous step as input to the synthesised architecture-level system representation (Section 5.2.3). The fourth step (*D* in Figure 5.2) employs the antipattern-based analysis method to analyse the synthesised system model and identify potential areas in the system's operational profile where requirements are violated (Section 5.2.4). The fifth and final step (*E* in Figure 5.2) guides refactoring actions on both internal components and third-party services of the system if needed (Section 5.2.5). Section 5.3 evaluates the integrated methodology using the FX system case study from Chapter 4 combined with Java source code obtained from GitHub. Finally, Section 5.4 provides a comparison to related work in this area and Section 5.5 summarises the chapter.

5.1 Motivating example

To illustrate the application of our integrated approach, we consider the software system introduced in Chapter 4. More information regarding the type of requests and the system's workflow can be found in Section 4.1. We assume that the software engineers have full control over the internal components of the system, and thus, can modify their source code when needed. The parameters of the system that are outside the control of its developers represent its operational profile. The ranges of values that the engineers consider for these parameters typically reflect the engineers' expectation about a particular deployment of the system.

Under the previous assumption, we create a scenario, in which we suppose that the source code of the internal "technical analysis" (TA) component includes an implementation of the `minimumPathSum` algorithm. We further suppose that this is the dynamic-programming implementation of the algorithm freely available on GitHub at https://github.com/TheAlgorithms/Java/. This method calculates the minimum path sum in a $M \times N$ matrix, where M and N refer to length and width of the input array, respectively. As shown in Figure 5.1, the method receives as input a two-dimensional array, and performs the calculations based on the following rules: a) moving from the top left corner to the down right corner, and b) moving one step down or right.

In the context of our motivating example, the Java implementation of the algorithm depicted in Figure 5.1, and in general shortest path algorithms, can be applied in currency trading applications to offer solutions and improve the trading patterns. This use of `minimumPathSum` by a potential TA component is demonstrated in the scenario below.

```
1  public static int minimumPathSum(int[][] grid) {
2      int m = grid.length, n = grid[0].length;
3      if (n == 0) {
4          return 0;
5      }
6      int[][] dp = new int[m][n];
7      dp[0][0] = grid[0][0];
8      int i = 0;
9      while (i < n - 1) {
10         dp[0][i + 1] = dp[0][i] + grid[0][i + 1];
11         i++;
12     }
13     i = 0;
14     while (i < m - 1) {
15         dp[i + 1][0] = dp[i][0] + grid[i + 1][0];
16         i++;
17     }
18     i=1;
19     while (i < m) {
20         int j=1;
21         while (j < n) {
22             dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) +
             ↪ grid[i][j]; // @time=0.005
23             j++;
24         }
25         i++;
26     }
27     return dp[m - 1][n - 1];
28 }
```

Fig. 5.1: Java method `minimumPathSum` for calculating the minimum path sum in a $M \times N$ matrix.

Let c_1, c_2,..., c_n be n different currencies. For instance, c_1, c_2, and c_3 are euros, pounds, and dollars, respectively. The exchange rate $r_{x,y}$ between any two currencies c_x and c_y means that purchasing $r_{x,y}$ units of currency c_y requires exchanging one unit of currency c_x. Additionally, the exchange rate $r_{x,y}$ satisfies the condition that $r_{x,y} \cdot r_{y,x} < 1$, in the sense that starting with a unit of currency c_x, changing it into currency c_y, and then converting back the unit(s) to currency c_x, results into having less than one unit of currency c_x. This difference is due to the cost of the transactions. Given a set of exchange rates $r_{y,x}$ and two currencies c_x and c_y, a trader is able to find the most advantageous sequence of currency exchanges for converting currency c_x into currency c_y using the functionalities offered by the FX system.

Since we could not obtain the code of the TA component, we based our example on the following assumptions. The TA component could potentially consist of several different classes and Java methods. In our example, we focus on `minimumPathSum` which we suppose to be one of the TA component's methods and also the most compute-

intensive part of it. In addition, we assume that changes in other parts of the TA component do not offer any significant improvements to the component's performance or cause the violation of performance requirements at system level. Thus, the focus of the software engineers is to assess the performance of this method in case of a violation in any of the TA component's performance requirements.

Another method that is part of TA is `search`. Its impact will be explored later in Section 5.3.2 as it is not included in the initial performance assessment of TA, conducted by the engineers of the system.

We suppose that a detailed log reflecting the method's usage profile is available, i.e., the typical combinations of matrices that `minimumPathSum` is invoked with. Also, we suppose that engineers want to assess the method's expected execution time, if each execution of the statement from line 22 requires 0.005s on average. The annotation '@time=0.005' appended as comment to line 22 is used to specify the performance property whose evaluation is of interest.

5.2 Approach

To allow the integration of the two types of analysis, we developed a combined methodology that carries out performance analysis and refactoring both at code-level and at system architecture-level. The essence of this joint analysis is to a) enable the combined refactoring actions mentioned previously and b) give the software engineers the flexibility of tackling performance issues from different perspectives. No additional changes have been made to the two techniques, instead connection points have been introduced to facilitate the transitioning between the steps of the approach.

The steps of this methodology are depicted in Figure 5.2. The blue (step B) and yellow (step D) coloured steps indicate the analysis techniques presented in the previous two chapters of the thesis. The grey-coloured steps (A, C and E) are new steps required for the integration of the two methods . The two outgoing dotted red lines from step E are optional actions that are only undertaken if refactoring is required. These actions allow software performance engineering to be applied to both internal components and external services of the system, enabling code-level and system architecture-level refactoring for the internal components, and system architecture-level refactoring for the external services. The five steps of our approach are described in the following sections.

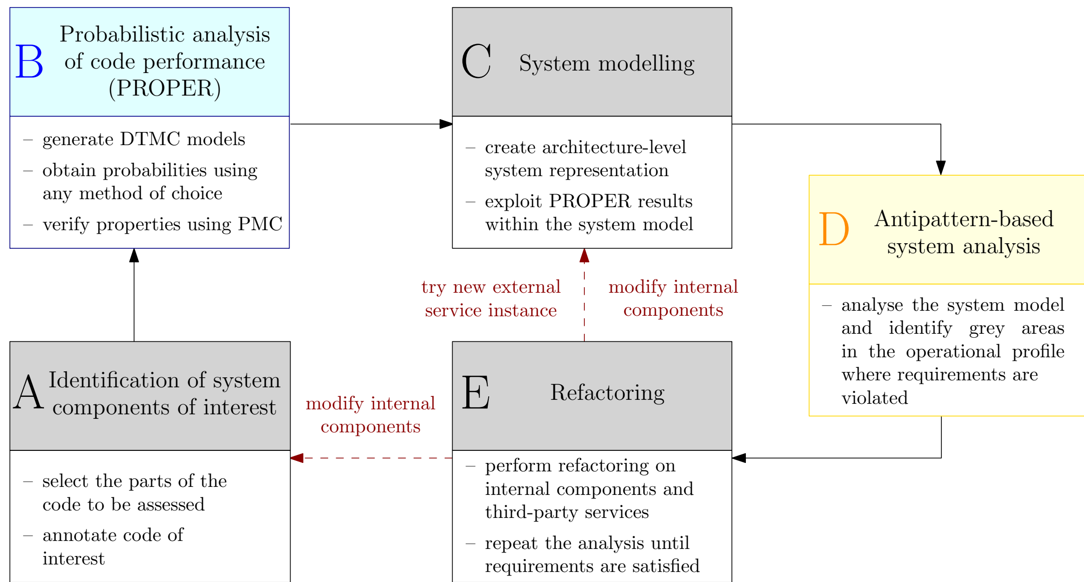


Fig. 5.2: Overview of the integrated approach for software performance engineering

5.2.1 Identification of System Components of Interest

In step A, the software engineer identifies internal components of interest from the system under analysis, and selects parts of the source code to be assessed. The selection of parts of the source code to be assessed is a process to be performed by the respective system's engineers and requires knowledge of the system's components and functionalities. As a general rule, the selection should focus on parts of the source code that are computationally intensive, and are responsible for most of the system's functionalities. In addition, attention should also be given to parts of the code that require the invocation of external functions in order to complete the tasks of the software system.

The assessment includes the analysis of performance properties, specified using annotations in the code. These annotations capture performance characteristics and are appended as comments (`//@property=value`) after the Java statements that these performance properties are associated with. For instance, consider line 22 in our motivating example (Figure 5.1) where a *time* value of 0.005 is associated with the invocation of *Math.min* function (`//@time=0.005`).

5.2.2 Probabilistic Analysis of Code Performance

Step B of the approach uses our probabilistic program performance analysis (PROPER) method, introduced in Chapter 3, to assess the annotated performance properties. This step includes the extraction of a discrete-time Markov chain (DTMC) model from (Java) source code (Section 3.2.1), the calculation of transition probabilities between states in

the DTMC model (Section 3.2.2), and the verification of properties using probabilistic model checking (Section 3.2.3).

PROPER receives as input the annotated Java code from step A and generates a reward-augmented DTMC model. The automated model extraction process is followed by the transition probability calculation for the model states corresponding to conditional statements and loops in the code. This calculation is performed based on the usage profile of the analysed code, obtained from program logs, under the assumption that the code is appropriately instrumented to generate logs containing this information.

Finally, the performance properties of interest, specified in PCTL, are verified by applying probabilistic model checking to the DTMC model. For this analysis, the DTMC transition probabilities are set to the previously calculated probabilities.

5.2.3 System Modelling

In step C, the performance model representing the entire system is manually developed to enable the analysis and computation of performance indices. Section 4.3 in Chapter 4 presents existing work on automating the above process. The effort required to develop and analyse these performance models depends on the complexity of the application domain and the expertise of the analysts. Our approach is not prescriptive about the type of performance models that can be used for system modelling. The only requirement is that the selected models must be able to capture the uncertainty associated with the operational profile of the system. As also mentioned in Chapter 4, our experiments use discrete-time and continuous-time Markov chains (DTMCs and CTMCs) for this part of the approach, with the purpose of capturing uncertainty.

The results of step B, obtained using PROPER, are then used within the system architecture-level model, together with the rest of the system parameters. This step involves the instantiation of the devised models as in Section 4.2.2, with the difference that now the internal components can be analysed using PROPER to obtain the associated parameter values. This introduces more options for the software engineers towards ensuring the system's optimal performance.

5.2.4 Antipattern-based System Analysis

Step D of the approach applies our antipattern-based performance analysis method from Chapter 4 to the system model, and identifies areas in the system's operational profile where requirements may potentially be violated.

The antipattern-based analysis method receives as input the performance models developed in setp C and computes the performance indices corresponding to all considered combinations of operational profile parameter values (see Section 4.2.3). The resulting performance indices and a portfolio of antipattern detection rules are then used to identify the performance antipatterns that occur for different combinations of parameter values. As described in Section 4.2.4, this method produces a series of maps that depict the distribution of antipatterns across the system's entire operational profile space, and highlight areas where performance violations occur. The occurrence of performance antipatterns indicates whether refactoring actions are required, e.g., in regions of the operational profile space where the deployed system is expected to operate.

5.2.5 Refactoring

Suitable refactoring actions are then considered in the final step to satisfy any requirements that have been violated. The refactoring actions can target the system components both at system architecture-level (e.g., trying a new external service instance), and at code-level (e.g., replacing a function in the source code). The analysis and refactoring processes are repeated until the system operation is satisfactory from a performance viewpoint.

One of the main benefits of the integrated approach is the choice between refactoring actions. A portfolio of such actions is provided from the antipattern-based system analysis method, which focuses on system architecture-level, and can be adapted to support additional case studies. At the same time, PROPER's application and analysis on the internal components' source code can guide the software engineers towards appropriate refactoring actions at code-level. These code-level refactoring actions may include the use of approximate functions (e.g., replacing a function with a functionally equivalent but faster function) or the restructuring of code (e.g., reordering an else-if statement that includes a performance-wise costly operation in the condition of the first if-statement and is rarely satisfied, by changing the order of the if-statements so that other conditions are evaluated first).

The flexibility of performance assessment at both system architecture-level and code-level can lead to efficient system performance optimisation, minimizing the potential cost. For instance, replacing a function in the target internal component's source code with a faster alternative is less costly than introducing a second thread at system level of that component, while both options satisfy the performance requirements.

This combination of refactoring actions provides better control over the system, and empowers engineers to tackle performance violations from different perspectives.

5.3 Evaluation

In this section, we first introduce the research questions that we intend to address (see Section 5.3.1). Thereafter, we describe the experimental scenarios (see Section 5.3.2) and discuss the obtained results (see Section 5.3.3). We finally report the threats to validity in Section 5.3.4.

5.3.1 Research Questions

We evaluated the integrated approach by performing combined experiments to answer the following research questions.

- RQ_1: Does the integrated approach guide refactoring actions at code-level through system architecture-level performance analysis?
- RQ_2: Does PROPER provide results that can be meaningfully used at system architecture-level for a combined performance analysis?
- RQ_3: Does the integrated approach provide insights on the performance of a software system using the combined code-level and system architecture-level performance analysis?

In order to answer these questions, we applied the integrated approach to the FX system from Chapter 4 (Section 4.1) and the code introduced in Sections 5.1 and 5.3.2 of this chapter.

5.3.2 Experimental Scenarios

The application of our integrated approach starts with annotating the code of interest (step A), and proceeding to analyse it using PROPER (step B). Figure 5.3 shows the DTMC obtained by applying PROPER to the Java code from Figure 5.1. The statement modelled by each of the 21 DTMC states is mentioned under the state.

Figure 5.4 shows the PRISM-encoded DTMC model generated by PROPER for the `minimumPathSum` Java method in our case study. The model has one reward structure, corresponding to the `time` annotation from the Java code in Figure 5.1. The transition probabilities `p1` and `p2`, `p3`, `p4`, `p5` correspond to the ‘if’ statement and

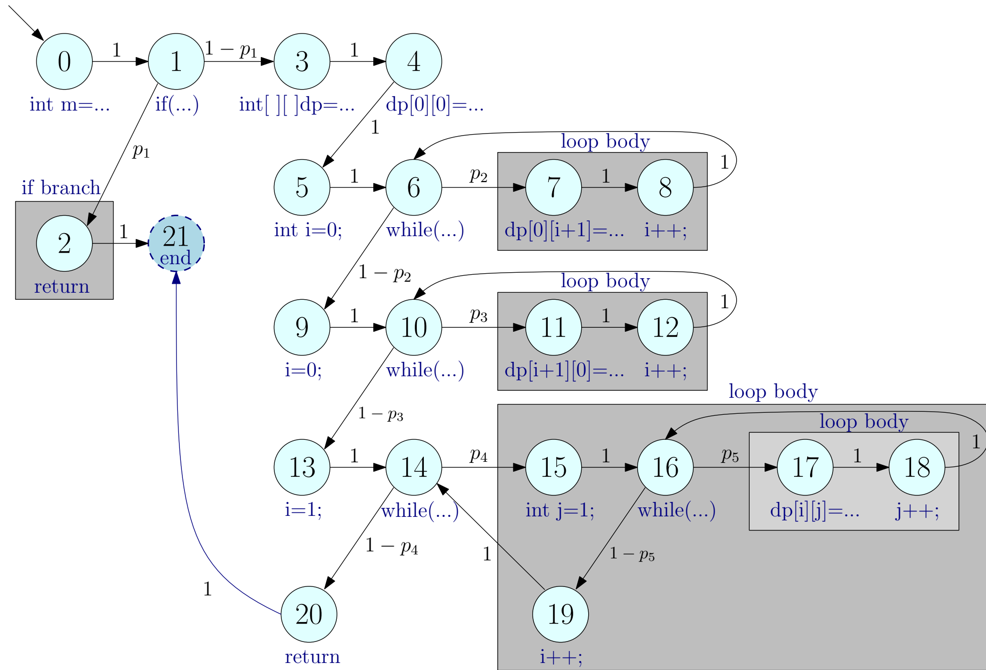


Fig. 5.3: DTMC model for `minimumPathSum` Java method.

'while' loops from the Java code, respectively. Their values depend on the usage profile of the code, and are determined as described in Chapter 3.

Using a log that captures the method's usage profile, we can extract the probabilities of executing different branches within the code. By adding these probabilities to the DTMC model, we can quantify the 'time' property through probabilistic model checking of the reward PCTL property $R\{\text{“}time\text{”}\} = ?[F\ s = end_state]$ over the DTMC model from Figure 5.4. In our experimental study, we used a synthetic-data log that was automatically populated with randomly generated input data for the `minimumPathSum` method. Using this log, the result of the PROPER analysis is $time = 0.0919s$.

In step C, we combine the result obtained via the PROPER analysis with the system model. The system architecture-level model of this case study uses the rate of each internal component and third-party service as parameters. To calculate the execution rate of the TA component, first we must obtain all 'time' property values associated with each of the parts of TA, as done for `minimumPathSum` method of our example above. Then by adding up all the values obtained via PROPER analysis on each part of

```
dtmc

const double p1;
const double p2;
const double p3;
const double p4;
const double p5;
const int end_state = 21;

module minimumPathSum
  s : [0..end_state] init 0;

  [] s=0  -> 1:(s'=1);                   //line:2
  [] s=1  -> p1:(s'=2)+(1-p1):(s'=3);    //line:3
  [] s=2  -> 1:(s'=end_state);           //line:4
  [] s=3  -> 1:(s'=4);                   //line:6
  [] s=4  -> 1:(s'=5);                   //line:7
  [] s=5  -> 1:(s'=6);                   //line:8
  [] s=6  -> p2:(s'=7)+(1-p2):(s'=9);    //line:9
  [] s=7  -> 1:(s'=8);                   //line:10
  [] s=8  -> 1:(s'=6);                   //line:11
  [] s=9  -> 1:(s'=10);                  //line:13
  [] s=10 -> p3:(s'=11)+(1-p3):(s'=13);  //line:14
  [] s=11 -> 1:(s'=12);                  //line:15
  [] s=12 -> 1:(s'=10);                  //line:16
  [] s=13 -> 1:(s'=14);                  //line:18
  [] s=14 -> p4:(s'=15)+(1-p4):(s'=20);  //line:19
  [] s=15 -> 1:(s'=16);                  //line:20
  [] s=16 -> p5:(s'=17)+(1-p5):(s'=19);  //line:21
  [] s=17 -> 1:(s'=18);                  //line:22
  [] s=18 -> 1:(s'=16);                  //line:23
  [] s=19 -> 1:(s'=14);                  //line:25
  [] s=20 -> 1:(s'=end_state);           //line:27
  [] s=21 -> 1:(s'=21);
endmodule

rewards "time"
  s=17 : 0.005;
endrewards
```

Fig. 5.4: PRISM model synthesised for the `minimumPathSum` Java method.

TA we can calculate TA's rate as follows:

$$TA_{time} = TA_{time_1} + TA_{time_2} + \cdots + TA_{time_n}$$

$$TA_{time} = TA_{time_{minPathSum}} + TA_{time_{otherParts}}$$

$$TA_{time} = 0.0919 + 0.1839$$

$$TA_{time} = 0.2758$$

$$TA_{rate} = \frac{1}{TA_{time}}$$
$$TA_{rate} = \frac{1}{0.2758}$$
$$TA_{rate} = 3.625$$

where $TA_{time_1} \ldots TA_{time_n}$ refer to the obtained 'time' property results for each of the parts of TA, $TA_{time_{minPathSum}}$ is the time value for `minimumPathSum`, and we suppose that $TA_{time_{otherParts}}$ represents the added time value for the other parts of TA.

We can now apply the antipattern-based performance analysis to the complete system model and identify potential areas of requirement violation (step D). Figure 5.7 shows the automatically generated maps for identifying "BLOB" antipattern occurrences in different operational profile scenarios. This antipattern characterises components that *"perform most of the work of the system, relegating other classes to minor, supporting roles"* [144]. If the system operates in areas of the operational profile where antipatterns appear, software engineers can then apply suitable refactoring actions to remedy these issues (step E).

The maps obtained for the initial configuration of the system (Figure 5.7) indicated the need for refactoring actions for the TA component. To demonstrate another application of our approach at code-level, we consider a scenario in which the engineers of the system, instead of improving the response time of `minimumPathSum` method that the TA component uses, seek to improve the response time of the `search` method (also part of the TA component), briefly introduced in Section 5.1.

This method implements a *binary search* algorithm, i.e., finds the position of a target value within a sorted array. We suppose that the implementation of this method is the one available freely on GitHub at https://github.com/TheAlgorithms/Java/. An example of the method's use in the FX system would be to obtain the position of a value, corresponding to the exchange rate of a currency, from a sorted list containing all the exchange rates of that currency towards all other currencies.

In this scenario, the improvement in `search`'s response time could be achieved by code restructuring. A procedure that not only improves the code's efficiency but also is less costly than invoking a faster but potentially more expensive external function, e.g., as in the case of `minimumPathSum`. The Java source code of `search` method along with the respective generated DTMC model, before restructuring has been applied, can be seen in Figures 5.5 and 5.6, respectively.

Figure 5.5 depicts the respective Java code consisting of multiple 'if' and 'if-else' statements, some of which contain function invocations in their conditions (lines 9

```
1  public class BinarySearch {
2
3     public int search(int array[], int key, int left, int right) {
4        if (right < left) {
5           return -1;
6        }
7        int median = (left + right) >>> 1;
8
9        if (Integer.compare(key,array[median]) == 0) {
          ↪ //@time=0.0238
10          return median;
11       }
12       else {
13          if (Integer.compare(key,array[median]) < 0) {
             ↪ //@time=0.0238
14             return search(array, key, left, median - 1);
15          }
16          else {
17             return search(array, key, median + 1, right);
18          }
19       }
20    }
21 }
```

Fig. 5.5: Java implementation of the search method

and 13). The associated ‘time’ property value for each of these statements is $0.0238s$. As each function invocation that checks whether the condition of the ‘if’ statement is satisfied takes $0.0238s$, it would be efficient performance-wise to arrange the statements starting with the one whose condition is more likely to be satisfied. This can be determined using historical data logs for the search method, as previously done for minimumPathSum.

Using a synthetic log that captures the method’s usage profile, we were able to obtain the probabilities of executing the different branches within the code. These probabilities were used to both quantify the ‘time’ property through probabilistic model checking of the reward PCTL property $R\{``time"\} = ?[F\ s = end_state]$ over the DTMC model from Figure 5.6, and to assess the likelihood of executing each of the ‘if’ conditions. The result of PROPER analysis for this method is $TA_{time_{search}} = 0.0918s$ (included in $TA_{time_{otherParts}}$), and corresponds to the response time before the mentioned refactoring. The results for this method after the applied refactoring can be seen in the following section.

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6
7  const int end_state = 8;
8
9  module binarySearch
10   s : [0..end_state] init 0;
11
12   [] s=0 -> p1:(s'=1)+(1-p1):(s'=2);  //line:4
13   [] s=1 -> 1:(s'=end_state);         //line:5
14   [] s=2 -> 1:(s'=3);                 //line:7
15   [] s=3 -> p2:(s'=4)+(1-p2):(s'=5);  //line:9
16   [] s=4 -> 1:(s'=end_state);         //line:10
17   [] s=5 -> p3:(s'=6)+(1-p3):(s'=7);  //line:13
18   [] s=6 -> 1:(s'=0);                 //line:14
19   [] s=7 -> 1:(s'=0);                 //line:17
20   [] s=8 -> true;
21 endmodule
22
23 rewards "time"
24   s=3 : 0.0238;
25   s=5 : 0.0238;
26 endrewards
```

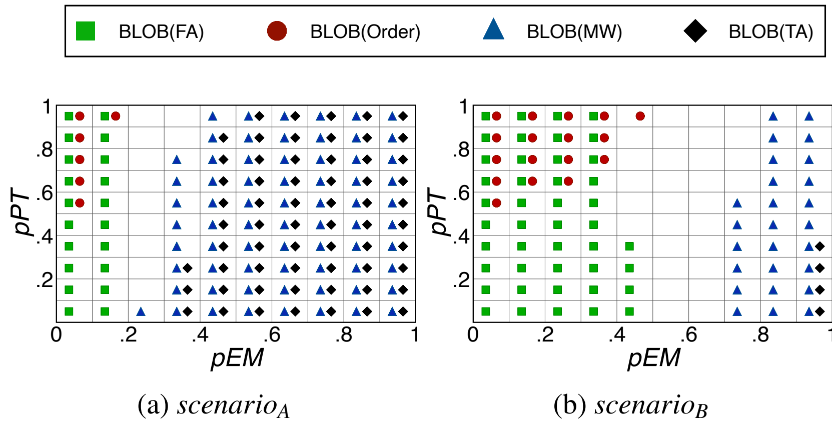
Fig. 5.6: DTMC model representation of the `search` Java method

5.3.3 Results and Discussion

The operational profile space of our adapted heterogeneous software system is fully defined by the following branching point probabilities: (i) the probability of executing the workflow in expert mode (p_{EM}); (ii) the probability of successfully performing a transaction (p_{PT}); (iii) the probabilities of satisfying the objectives (p_{OS}) or not (p_{ON}). As such, the resulting probability of an error occurring is $1-(p_{OS}+p_{ON})$.

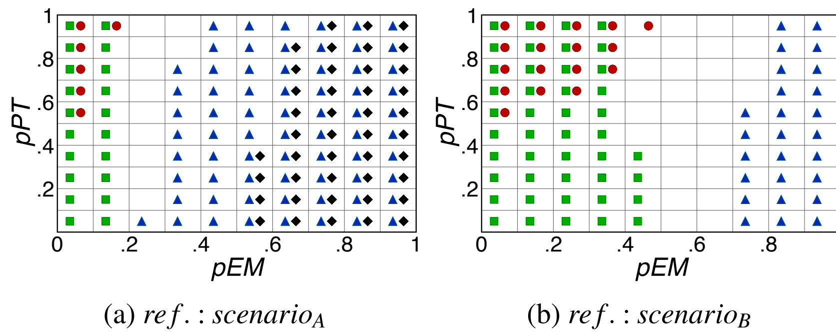
The experimental scenarios analyzed in this section vary p_{EM} and p_{PT} within their full range $[0,1]$ with a 0.1 step. For p_{OS} and p_{ON} we use two combinations of values, i.e., $(0.21, 0.78)$ and $(0.48, 0.01)$, which we refer to as $scenario_A$ and $scenario_B$, respectively.

We present results obtained using the automated tool and detection rules that we introduced in Chapter 4. The tool generates maps of antipattern occurrences across the entire operation profile space (depicted as coloured shapes in the maps). The antipattern occurrences for each system component appear in the general form: $antipattern_name(component_name)$ (e.g., BLOB(Order)).



(a) $scenario_A$ (b) $scenario_B$

Fig. 5.7: BLOB antipattern instances while varying operational profiles.



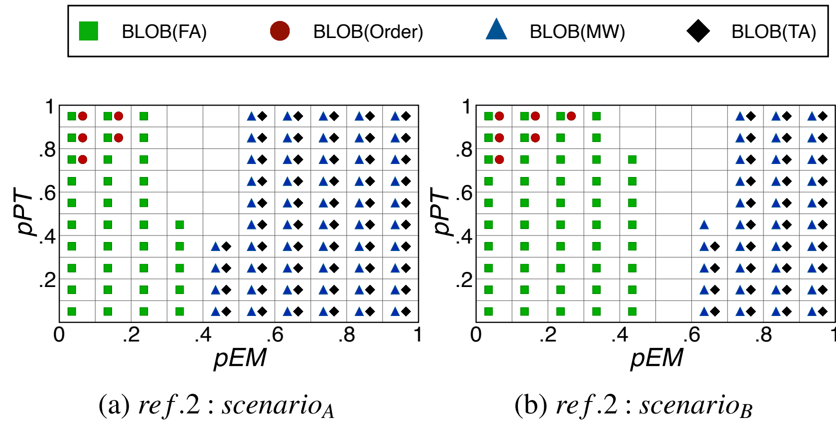
(a) $ref.: scenario_A$ (b) $ref.: scenario_B$

Fig. 5.8: BLOB antipattern instances when improving the rate of the TA component (code-level refactoring).

Figures 5.7a and 5.7b show the antipattern occurrences in two different operational profile scenarios ($scenario_A$ and $scenario_B$) for the system with $TA_{rate} = 3.625$. To reduce the occurrences of the BLOB(TA) antipattern in the region of the operational profile where the system is expected to operate, the software engineers can introduce a faster function in the code (Figure 5.1 line 22) $FastMath.min$ that requires 0.002s on average instead of 0.005s. This improvement (which is not an architectural change, but was identified through architectural-level analysis) results in a reduction in the expected time of `minimumPathSum` ($time = 0.0367s$), and an increase in the overall rate of the TA component:

$$TA_{rate} = \frac{1}{0.0367 + 0.1839}$$
$$TA_{rate} = \frac{1}{0.2206}$$
$$TA_{rate} = 4.53$$

Figures 5.8a and 5.8b show the decreased BLOB(TA) antipattern occurrences after



(a) $ref.2 : scenario_A$ (b) $ref.2 : scenario_B$

Fig. 5.9: BLOB antipattern instances when introducing a second thread of the TA component (architecture-level refactoring).

the applied refactoring action.

> *Summary for RQ_1:* The above findings provide the answer to RQ_1 from Section 5.3.1 by showing that our approach enables the identification of suitable refactoring actions at code-level through system architecture-level performance analysis.

Figures 5.9a and 5.9b show the antipattern occurrences in $scenario_A$ and $scenario_B$ after introducing a second thread of the TA component to the already refactored system (Figures 5.8a and 5.8b). This second thread is added to split the incoming load and manage users' requests, reducing their drop rate given that the queue length remains unchanged. However, in practice it does not relieve TA itself and that is the reason for BLOB(TA) to not disappear. More BLOB(TA) instances appear after this refactoring as more user requests are being handled. At the same time, a better balance between the two system branches has been achieved with now less BLOB(MW) and BLOB(order) instances and more BLOB(FA) instances being depicted in Figures 5.9a and 5.9b.

The option of combining architecture and code-level refactoring actions enhances the control that software engineers have over the system towards the satisfaction of its performance requirements, and it is up to them to decide which refactoring action best suits the system's needs.

> *Summary for RQ_2:* The above findings provide the answer to RQ_2 by showing that our approach enables a combined performance analysis using the two mentioned approaches. Specifically, PROPER's results, obtained at code-level, are used within the system architecture-level model, and after applying the antipattern-based approach, the engineers are informed whether any of the system's requirements has

been violated and can apply suitable refactoring actions at any level (as seen in the two cases described above).

From the previous experiments we obtained $TA_{time_{minPathSum}} = 0.0919s$ for the response time of the `minimumPathSum` Java method, $TA_{time_{search}} = 0.0918s$ for the response time of the `search` method, and given the $TA_{time_{otherParts}} = 0.0921s$ as the assumed response time for the remaining parts of TA, we obtain $TA_{time} = 0.2758s$ as the overall response time of the TA component for the initial system configuration.

After the PROPER analysis of the `search` method (Figure 5.5), we identified that the 'if' statement in line 13 and the 'else' statement in line 16 have higher probabilities of execution than the earlier 'if' statement in line 9. This observation is interpreted based on the probabilities obtained from the log capturing the method's usage. The current structure of the method results into additional execution time as the *compare* function in the condition of the 'if' statement (line 9) will be invoked multiple times without the condition being satisfied. This could be prevented by applying code restructuring and improve the performance of the method.

The applied changes can be observed (highlighted) in Figures 5.10 and 5.11 depicting the Java code and the resulting DTMC model, respectively. The 'if' statement at line 9 in Figure 5.5 is now moved at the end of the 'if-else' block at line 16 in Figure 5.10 as its condition has the lowest probability of being satisfied. The result of PROPER analysis for the updated `search` method is only $TA_{time_{search}} = 0.0737s$. Accordingly, the new TA_{rate} achieved after this restructuring is given by the following calculations:

$$TA_{time} = TA_{time_{minPathSum}} + TA_{time_{search}} + TA_{time_{otherParts}}$$
$$TA_{time} = 0.0919 + 0.0737 + 0.0921$$
$$TA_{time} = 0.2577$$

$$TA_{rate} = \frac{1}{0.2577}$$
$$TA_{rate} = 3.88$$

The generated maps of antipattern occurrences for the system with $TA_{rate} = 3.88$ (concerning $scenario_A$ and $scenario_B$) are identical to the ones from Figures 5.8a and 5.8b. This means that, without improving the response time of `minimumPathSum` method (i.e., just by performing code restructuring in `search` method), we were able to

```java
1  public class BinarySearch {
2
3      public int search(int array[], int key, int left, int right) {
4          if (right < left) {
5              return -1;
6          }
7          int median = (left + right) >>> 1;
8
9          if (Integer.compare(key,array[median]) < 0) {
           ↪ //@time=0.0238
10             return search(array, key, left, median - 1);
11         }
12         else {
13             if (Integer.compare(key,array[median]) > 0){
               ↪ //@time=0.0238
14                 return search(array, key, median + 1, right);
15             }
16             else {
17                 return median;
18             }
19         }
20     }
21 }
```

Fig. 5.10: Java implementation of the `search` method after code restructuring has been applied

increase the overall rate of the TA component up to a level that satisfies the performance objectives of the system without increasing the potential cost.

Of course, in a scenario in which the system's performance objectives were not satisfied from the above refactoring action, the engineers would need to continue to assess the impact of the requirement(s) violation. Such an assessment may lead to the engineers deciding to proceed with improving the `minimumPathSum` to achieve a better rate for the TA component, or to keep the potential cost as well as the rate at a lower level. The latter option would be particularly appropriate if the antipattern occurrences are in regions of the operational profile where the system is less likely to operate.

> *Summary for RQ*3*:* The findings throughout Section 5.3.3 show that our integrated methodology can provide insights on the performance of a software system using a combination of both levels of performance analysis (code and system architecture-level).

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6
7  const int end_state = 8;
8
9  module binarySearch
10   s : [0..end_state] init 0;
11
12   [] s=0 -> p1:(s'=1)+(1-p1):(s'=2);   //line:4
13   [] s=1 -> 1:(s'=end_state);          //line:5
14   [] s=2 -> 1:(s'=3);                  //line:7
15   [] s=3 -> p2:(s'=4)+(1-p2):(s'=5);   //line:9
16   [] s=4 -> 1:(s'=0);                  //line:10
17   [] s=5 -> p3:(s'=6)+(1-p3):(s'=7);   //line:13
18   [] s=6 -> 1:(s'=0);                  //line:14
19   [] s=7 -> 1:(s'=end_state);          //line:17
20   [] s=8 -> true;
21 endmodule
22
23 rewards "time"
24   s=3 : 0.0238;
25   s=5 : 0.0238;
26 endrewards
```

Fig. 5.11: Updated DTMC model representation of the `search` Java method

5.3.4 Threats to Validity

Construct validity. Threats may arise from the simplifications and assumptioms made when constructing the parametric DTMC and CTMC models for the selected Java code and the architecture of the software system from the case study. To mitigate these threats, we devised these models using our peer-reviewed methods from [28] and [152], and employed established software performance engineering simplifications (e.g., assuming exponentially distributed request arrival rates).

Internal validity. Threats can originate from obtaining inaccurate results via simulating the code's execution, and can also be associated with errors resulting from the detection of potentially spurious performance antipatterns. To mitigate the first type of threats, we performed simulation up to 10^4 times. Additionally, we created 10 sets of these simulation runs and calculated the average of their output values. Regarding the second type of threats, we verified that the detected performance antipatterns follow the given rules defined in their specification, along with the expected performance indicators as in Section 4.4.4.

External validity. Threats arise due to the difficulty of applying the proposed methodology to other software systems than the one from our case study. To mitigate this threat, we based our case study on a component-based software system taken from the research literature and used by mutliple other projects, and we applied the PROPER step of the methodology to Java methods taken from existing software libraries. Nevertheless, additional case studies and experiments are needed to confirm the generality of our methodology.

5.4 Related Work

Existing approaches focusing on the performance analysis of software systems at both code-level (e.g. [4], [11], [103]) and architecture-level (e.g. [1], [6], [74]) do exist, and have been explored in the previous chapters. However, to the best of our knowledge, there is no work in the literature that aims for an integrated performance analysis at both levels, which is achieved by our approach. Additionally, there is no architecture-level approach that focuses on how the operational profile affects the performance analysis and refactoring of software systems, and the idea of employing performance antipatterns shows promising results according to our experimental results in both Sections 4.4 and 5.3.

The approach introduced in [68] enables reliability modelling and analysis for component-based systems. Additionally, it allows dealing with multiple failure modes, studying the error propagation among components, and ensuring an optimal combination of component reliability properties' values. This approach differs from our methodology as it focuses on reliability modelling and analysis. However, we see the benefit of a potential integration of this approach into the solution introduced in this chapter, as a way to enable the analysis of reliability properties by our methodology.

In [36] a decentralized architecture is proposed for building a fully functional assembly of distributed services, not only able to optimize its energy consumption, but also paying attention to arising issues regarding the delivered quality of service. This approach differs from ours as it lies within the area of service selection and composition in a distributed environment.

Focusing on the analysis of nonfunctional attributes of component-based systems, [84] defines a model-driven transformation framework based on a kernel language that captures relevant information, enabling this type of analysis. This approach brings together design-oriented and analysis-oriented notations and reduces the burden of defining a variety of direct transformations between notations with the kernel language

acting as an intermediary. While this approach is limited at architectural level compared to ours, it offers a solution to the problem of automating the transformation of design models to analysis models.

5.5 Summary

We presented an integrated approach for the verification of software performance properties at both code-level and system architecture-level, that allows software engineers to perform suitable refactoring actions when a performance requirement violation occurs. PROPER is used for synthesising a DTMC model using code annotated with performance properties of interest and quantifying these properties by applying probabilistic model checking. The obtained results are combined with a system architecture-level model and analysed by the antipattern-based system analysis method to identify regions of the system's operational profile where antipatterns occur, and drive suitable refactoring actions. We applied this approach in a case study, and showed how it can be used to assist software engineers maintaining the performance requirements set at both system and code-level.

Chapter 6

Conclusion and Further Research Directions

6.1 Conclusion

This thesis highlighted the importance of performance analysis during the development process of a software system, and addressed several limitations of existing solutions. Analysing software performance at code-level, carried out by existing techniques is a tedious and time consuming process that needs to be repeated for every considered platform, usage profile of interest and after every code change. Existing system architecture-level performance analysis techniques have also restricted application due to the fact that they operate under the assumption that the system's operational profile is known and does not change over time. Considering these limitations, we defined the following research hypothesis:

> *Given the formalisation of the architecture of a software system and/or of the source code of its relevant components as discrete-time Markov chains, and a set of nonfunctional requirements encoded in probabilistic temporal logic, probabilistic model checking combined with performance antipatterns can:*
>
> 1. *provide guarantees that these requirements are met for certain operational-profile regions;*
> 2. *guide the refactoring of the software system to ensure it meets the requirements for operational-profile regions of interest for which its initial version violates these requirements.*

The two-pronged approach that we devised enables the analysis and refactoring of software systems that must comply with strict nonfunctional requirements. Unlike previous solutions, this approach operates at both code level and system architecture level.

To support performance analysis at code-level, we developed PROPER, a tool-supported method for the formal analysis of timing, resource use and other quality aspects of computer programs. PROPER synthesises a Markov chain model of the analysed code, computes its transition probabilities using information from program logs, and employs probabilistic model checking to evaluate the performance properties of interest. Additionally, the Markov chain model transformation step of PROPER has been automated to assist engineers by eliminating potential errors occurring from manually transforming the code into a Markov chain model. We demonstrated how quality attributes can be first identified in the source code of interest and then established using code fragments obtained from real-world applications. Our experimentation showed the benefits of using PROPER compared to simulating the execution of the code, in terms of accuracy, decision-making and efficiency.

For system architecture-level performance analysis, we devised an antipattern-based method that employs performance antipatterns and stochastic modelling to support refactoring. This method computes the performance antipatterns present across the operational profile space of a software system, enabling engineers to detect operational profiles likely to be problematic for the analysed design, and supporting the selection of refactoring actions when performance requirements are violated. We introduced the concept of *Performance Antipattern Profile*, which is a representation of performance antipattern occurrences while varying the operational profile parameters, and showed how a portfolio of refactoring actions can be defined for both internal components and external services. This portfolio can be used by the software engineers in selecting suitable refactoring actions when a performance requirement of the system has been violated. We demonstrated the application of the method in a scenario involving a foreign currency trading system comprising both in-house components and external services. The results of our experimentation highlighted the ability of the approach to identify problematic areas from a performance viewpoint in the system's operational profile and suggest suitable refactoring actions.

Finally, the two methods are integrated into an end-to-end software performance methodology that combines code-level probabilistic analysis with the use of performance antipatterns to guide refactoring. We illustrated the applicability of combining the proposed approaches using code obtained from various Java programs, includ-

ing Android applications, and the high-level design of a foreign currency exchange service-based system.

6.2 Further Research Directions

The research presented in this thesis can potentially be refined and extended in multiple directions. In this chapter we present those opportunities for improvement and expansion which we identified as the most promising research directions.

6.2.1 Probabilistic Analysis of Code Performance

We propose four directions of further research for the code-level analysis method introduced in the thesis: (1) extending PROPER to support analysis of reliability properties; (2) investigating methods to support the computation of confidence intervals of performance properties [27]; (3) applying PROPER to other applications and scenarios, and assessing its scalability to larger programs; and (4) validating PROPER in studies where it is used by practitioners. Each of these directions of further research is detailed next.

6.2.1.1 Reliability Properties

The current implementation of PROPER supports the analysis of performance properties. The type of supported properties can be expanded, starting with reliability. We have already performed an initial experimentation towards this direction and reliability properties can be defined using PCTL in the form $P =?[F\ s = \text{“}n\text{”}]$, where n represents a state of interest for the system's engineers. This PCTL property is translated in natural language as 'the probability of eventually reaching state n'. While the analysis of reliability properties is possible using PROPER's current state, it is only applicable when dealing with fragments of code that do not contain loops.

In the case of loops, restrictions apply that require further experimentation to generalise the verification of reliability properties using PROPER. Specifically, as also mentioned in Section 3.2.2, the reason for this is that the calculation of the probability of reaching a state n inside a loop is different than the probability of satisfying the loop's condition. An extension of the theoretical foundation used by the approach is required, so that DTMCs can be created that support the analysis of the unsupported property types.

6.2.1.2 Confidence Intervals for Performance Properties

Another direction of further research is the computation of confidence intervals for the performance properties identified in source code using PROPER. In preliminary experimentation, we incorporated FACT [30] into PROPER's workflow, replacing the transition probability calculation step with the computation of confidence intervals for the evaluated properties.

FACT is a probabilistic model checker that was built based on the concept of explicitly taking into account the estimation errors that can appear in quantitative verification [27]. Dealing with estimation errors becomes possible by computing confidence intervals for the evaluated properties of parametric DTMCs with unknown transition probabilities when observations of these transitions are available. As there is no longer a need to calculate the transition probabilities between the states of the model, only the counts of reaching each branch in the code need to be extracted. These counts can be obtained directly from program logs and be given as input to FACT along with the respective parametric DTMC model.

The result of a verifiable property depends on the information we have about the operational profile of the system, i.e., the number of entries in the log of the application. The more entries a log contains, the higher the confidence of results obtained via verification will be.

6.2.1.3 Scalability

In order to assess the scalability of the approach, it is necessary to apply it in larger programs and scenarios. The restrictions of PROPER's current implementation, summarised in Section 3.2, need to be first addressed to support additional scenarios and variations of source code identified in real-world applications. As these restrictions are only associated with the current implementation of the automated tool and not with the method, we are positive that PROPER can be used to evaluate larger programs as quantitative verification has shown its efficiency when dealing with larger models. Advances in this area targeting the scalability problem include the use of abstraction [67] and compositional verification [110], while a general overview is given in [108]. Further experimentation is necessary to address scalability concerns and it is definitely an important line of future work for our approach.

6.2.1.4 Practicality

Even though the current case studies cover a variety of domains (see Section 4.1.2), PROPER needs to be further evaluated through studies where it is used by practitioners.

In Section 3.2 we referred to Java application domains and examples of applications that PROPER can potentially be applied to in the future. Additionally, to expand on its practicality, PROPER needs to support more programming languages as Java is only one of the most prominent ones. An initial discussion on this matter is also provided in Section 3.2 where we explain that this extension could be achieved without a lot of effort.

6.2.2 Software System Analysis & Refinement Using Performance Antipattern Profiles

In addition to the areas of further research mentioned in Section 4.4.4, our approach for software analysis and refinement using performance antipattern profiles can be augmented with the ability to handle reliability and costs constraints, and thus to support trade-off analysis among multiple quality attributes. The applicability of the approach could also be extended by developing a portfolio of generic refactoring actions (which need to be feasible with our modelling and analysis techniques), and methods that automate the selection of suitable actions from this portfolio. Additional details about these research directions are provided in the following sections.

6.2.2.1 Trade-off Analysis

Supporting trade-off analysis by handling additional nonfunctional properties enables better control over the system's performance objectives, and provides alternative solutions to performance requirements' violations while, at the same time, ensuring the overall system's performance.

Research work focusing on trade-off analysis between nonfunctional requirements has already been established [52, 53, 55, 120, 136, 137] and can potentially be adapted by our approach to ensure the balanced operation of a software system under development. Specifically for handling reliability and cost quality attributes, [52] enables the selection of software components based on a cost/reliability trade-off, aiming at minimizing the cost of the whole assembly subject to reliability constraints. Trade-off analysis is not limited between the system's nonfunctional requirements, but work has also been done [127] towards the direction of enabling the trade-off between functional and nonfunctional system requirements.

6.2.2.2 Portfolio of Generic Actions and Methods

To assist software engineers or users of our approach, we identify the need of extending the applicability of the approach by devising a portfolio of generic refactoring actions,

and automate their selection. In the evaluation of the FOREX case study (Section 4.4.2), we established various refactoring actions as a solution to the occurrence of the considered performance antipatterns. The refactoring actions included replacing an internal component instance with a faster alternative, supplying the system with an additional thread to split the incoming load, and change the pattern of the external services. These actions can be generalised and associated with the occurrence of specific antipatterns, and thus, applied in other systems that exhibit similar antipattern occurrences in their operational profile space. For example, if an external service of the SUD is identified as a P&F antipattern, a possible generalised action would be to either change the service pattern or invoke a faster service instance. These actions could also be supported by the trade-off analysis, i.e., performing the suggested refactoring only when the other nonfunctional requirements such as cost remain satisfied.

6.2.3 Software Performance Engineering With Code-level Probabilistic Analysis and Performance Antipatterns

The further work directions for our integrated approach include its evaluation within additional case studies and automating steps of the approach that still require manual effort. The following sections contain information on how these research directions can be pursued.

6.2.3.1 Additional Case Studies

The aim of this further research direction is to showcase the applicability of the approach in different contexts, and provide more examples of how the connection between code-level and system architecture-level performance objectives can assist in optimising the overall system's performance. We have identified a potential case study in [78] (Happy Hour Organiser) that can be adopted to further evaluate the integrated methodology presented in this thesis. Also, more examples showing the interdependence between the code-level and system architecture-level analysis are needed to evaluate the applicability of the integrated methodology in various performance scenarios.

6.2.3.2 Automation

The process of synthesising the probabilistic model representing a software system is non-trivial and error-prone if done manually. Thus, we highlight the importance of automating parts of the integrated methodology that require manual effort. Achieving complete automation will not only minimise the error factor, but at the same time will save a significant amount of time allocated for model synthesis and evaluation. Last

but not least, full automation would enable the use of the methodology within the control loop of self-adaptive and autonomous systems [58, 76, 168], e.g., to support the reconfiguration of these systems in response to changes in their environment.

Step C of the integrated methodology (as seen in Figure 5.2) is associated with the creation of the architecture-level system representation, and a candidate for automation. A detailed discussion on how architectural descriptions can be automatically translated into probabilistic models is given in Section 4.3. Another approach that could potentially be applicable in our methodology is [79], which enables the transformation of UML sequence diagrams to Markov models. We believe that with the integration of any of the methods, presented both in Chapter 4 and in this section, into our methodology or with the development of a similar method, the automation of this step can be achieved.

Appendix A

Chapter 3 - Supplementary Material

The BNF listed on Section A.1 defines the syntax rules of the subset of Java that our PROPER method can handle. The Java source code and parametric DTMC models listed on Sections A.2, A.3 and A.4 were used for the evaluation of PROPER method, introduced in Chapter 3. The Java code was given as input to the method and the DTMC models were devised as described in the probabilistic model extraction step (Section 3.2.1).

A.1 Java Subset Syntax Specification

Declarations

⟨*variable declarator*⟩ ::= ⟨*variable declarator id*⟩
  | ⟨*variable declarator id*⟩ = ⟨*variable initializer*⟩

⟨*variable declarator id*⟩ ::= ⟨*identifier*⟩ | ⟨*variable declarator id*⟩ ‘[’ ‘]’

⟨*variable initializer*⟩ ::= ⟨*expression*⟩ | ⟨*array initializer*⟩

⟨*method declaration*⟩ ::= ⟨*method header*⟩ ⟨*method body*⟩

⟨*method header*⟩ ::= ⟨*method declarator*⟩

⟨*method declarator*⟩ ::= ⟨*identifier*⟩

⟨*method body*⟩ ::= ⟨*block*⟩ | ‘;’

⟨*array initializer*⟩ ::= ‘{’ ⟨*variable initializers*⟩? , ? ‘}’

⟨*variable initializers*⟩ ::= ⟨*variable initializer*⟩ | ⟨*variable initializers*⟩ , ⟨*variable initializer*⟩

⟨*variable initializer*⟩ ::= ⟨*expression*⟩ | ⟨*array initializer*⟩

Blocks and Commands

⟨*block*⟩ ::= ⟨*block statements*⟩?

⟨*block statements*⟩ ::= ⟨*block statement*⟩ | ⟨*block statements*⟩ ⟨*block statement*⟩

⟨*block statement*⟩ ::= ⟨*local variable declaration statement*⟩ | ⟨*statement*⟩

⟨*local variable declaration statement*⟩ ::= ⟨*local variable declaration*⟩ ‘`;`’

⟨*local variable declaration*⟩ ::= ⟨*type*⟩ ⟨*variable declarators*⟩

⟨*statement*⟩ ::= ⟨*statement without trailing substatement*⟩
 | ⟨*if then statement*⟩ | ⟨*if then else statement*⟩ | ⟨*while statement*⟩ | ⟨*for statement*⟩

⟨*statement no short if*⟩ ::= ⟨*statement without trailing substatement*⟩
 | ⟨*if then else statement no short if*⟩ | ⟨*while statement no short if*⟩

⟨*statement without trailing substatement*⟩ ::= ⟨*block*⟩ | ⟨*empty statement*⟩
 | ⟨*expression statement*⟩ | ⟨*return statement*⟩

⟨*empty statement*⟩ ::= ‘`;`’

⟨*expression statement*⟩ ::= ⟨*statement expression*⟩ ‘`;`’

⟨*statement expression*⟩ ::= ⟨*assignment*⟩ | ⟨*preincrement expression*⟩
 | ⟨*postincrement expression*⟩ | ⟨*predecrement expression*⟩
 | ⟨*postdecrement expression*⟩ | ⟨*method invocation*⟩

⟨*if then statement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement*⟩

⟨*if then else statement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement no short if*⟩ ‘`else`’
 ⟨*statement*⟩

⟨*if then else statement no short if*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement no short if*⟩ ‘`else`’
 ⟨*statement no short if*⟩

⟨*while statement*⟩ ::= ‘`while`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement*⟩

⟨*while statement no short if*⟩ ::= ‘`while`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*statement no short if*⟩

⟨*statement expression list*⟩ ::= ⟨*statement expression*⟩
 | ⟨*statement expression list*⟩ , ⟨*statement expression*⟩

⟨*return statement*⟩ ::= ‘`return`’ ⟨*expression*⟩? ‘`;`’

Expressions

⟨*constant expression*⟩ ::= ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*assignment expression*⟩

⟨*assignment expression*⟩ ::= ⟨*conditional expression*⟩ | ⟨*assignment*⟩

⟨assignment⟩ ::= ⟨left hand side⟩ ⟨assignment operator⟩ ⟨assignment expression⟩

⟨left hand side⟩ ::= ⟨expression name⟩ | ⟨field access⟩ | ⟨array access⟩

⟨assignment operator⟩ ::= '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>='
  | '>>>=' | '&=' | '^=' | '|='

⟨conditional expression⟩ ::= ⟨conditional or expression⟩
  | ⟨conditional or expression⟩ '?' ⟨expression⟩ ':' ⟨conditional expression⟩

⟨conditional or expression⟩ ::= ⟨conditional and expression⟩
  | ⟨conditional or expression⟩ '||' ⟨conditional and expression⟩

⟨conditional and expression⟩ ::= ⟨inclusive or expression⟩
  | ⟨conditional and expression⟩ '&&' ⟨inclusive or expression⟩

⟨inclusive or expression⟩ ::= ⟨exclusive or expression⟩
  | ⟨inclusive or expression⟩
  | ⟨exclusive or expression⟩

⟨exclusive or expression⟩ ::= ⟨and expression⟩
  | ⟨exclusive or expression⟩ '^' ⟨and expression⟩

⟨and expression⟩ ::= ⟨equality expression⟩
  | ⟨and expression⟩ '&' ⟨equality expression⟩

⟨equality expression⟩ ::= ⟨relational expression⟩
  | ⟨equality expression⟩ '==' ⟨relational expression⟩
  | ⟨equality expression⟩ '!=' ⟨relational expression⟩

⟨relational expression⟩ ::= ⟨shift expression⟩
  | ⟨relational expression⟩ '<' ⟨shift expression⟩
  | ⟨relational expression⟩ '>' ⟨shift expression⟩
  | ⟨relational expression⟩ '<=' ⟨shift expression⟩
  | ⟨relational expression⟩ '>=' ⟨shift expression⟩
  | ⟨relational expression⟩ 'instanceof' ⟨reference type⟩

⟨shift expression⟩ ::= ⟨additive expression⟩
  | ⟨shift expression⟩ '<<' ⟨additive expression⟩
  | ⟨shift expression⟩ '>>' ⟨additive expression⟩
  | ⟨shift expression⟩ '>>>' ⟨additive expression⟩

⟨additive expression⟩ ::= ⟨multiplicative expression⟩
  | ⟨additive expression⟩ '+' ⟨multiplicative expression⟩
  | ⟨additive expression⟩ '-' ⟨multiplicative expression⟩

⟨multiplicative expression⟩ ::= ⟨unary expression⟩
  | ⟨multiplicative expression⟩ '*' ⟨unary expression⟩
  | ⟨multiplicative expression⟩ '/' ⟨unary expression⟩
  | ⟨multiplicative expression⟩ '%' ⟨unary expression⟩

⟨*cast expression*⟩ ::= ‘(’ ⟨*primitive type*⟩ ‘)’ ⟨*unary expression*⟩
  | ‘(’ ⟨*reference type*⟩ ‘)’ ⟨*unary expression not plus minus*⟩

⟨*unary expression*⟩ ::= ⟨*preincrement expression*⟩
  | ⟨*predecrement expression*⟩
  | ‘+’ ⟨*unary expression*⟩
  | ‘−’ ⟨*unary expression*⟩
  | ⟨*unary expression not plus minus*⟩

⟨*predecrement expression*⟩ ::= ‘−’ ⟨*unary expression*⟩

⟨*preincrement expression*⟩ ::= ‘++’ ⟨*unary expression*⟩

⟨*unary expression not plus minus*⟩ ::= ⟨*postfix expression*⟩
  | ‘~’ ⟨*unary expression*⟩
  | ‘!’ ⟨*unary expression*⟩
  | ⟨*cast expression*⟩

⟨*postdecrement expression*⟩ ::= ⟨*postfix expression*⟩ ‘−’

⟨*postincrement expression*⟩ ::= ⟨*postfix expression*⟩ ‘++’

⟨*postfix expression*⟩ ::= ⟨*primary*⟩
  | ⟨*expression name*⟩
  | ⟨*postincrement expression*⟩
  | ⟨*postdecrement expression*⟩

⟨*method invocation*⟩ ::= ⟨*method name*⟩ ‘(’ ⟨*argument list*⟩? ‘)’
  | ⟨*primary*⟩ . ⟨*identifier*⟩ ‘(’ ⟨*argument list*⟩? ‘)’
  | ‘super’ . ⟨*identifier*⟩ ‘(’ ⟨*argument list*⟩? ‘)’

⟨*field access*⟩ ::= ⟨*primary*⟩ ‘.’ ⟨*identifier*⟩ | ‘super’ . ⟨*identifier*⟩

⟨*primary*⟩ ::= ⟨*primary no new array*⟩ | ⟨*array creation expression*⟩

⟨*primary no new array*⟩ ::= ⟨*literal*⟩ | ‘this’ | ‘(’ ⟨*expression*⟩ ‘)’
  | ⟨*class instance creation expression*⟩ | ⟨*field access*⟩
  | ⟨*method invocation*⟩ | ⟨*array access*⟩

⟨*class instance creation expression*⟩ ::= ‘new’ ⟨*class type*⟩ ‘(’ ⟨*argument list*⟩? ‘)’

⟨*argument list*⟩ ::= ⟨*expression*⟩ | ⟨*argument list*⟩ , ⟨*expression*⟩

⟨*array creation expression*⟩ ::= ‘new’ ⟨*primitive type*⟩ ⟨*dim exprs*⟩ ⟨*dims*⟩?
  | ‘new’ ⟨*class or interface type*⟩ ⟨*dim exprs*⟩ ⟨*dims*⟩?

⟨*dim exprs*⟩ ::= ⟨*dim expr*⟩ | ⟨*dim exprs*⟩ ⟨*dim expr*⟩

⟨*dim expr*⟩ ::= ‘[’ ⟨*expression*⟩ ‘]’

⟨*dims*⟩ ::= ‘[’ ‘]’ | ⟨*dims*⟩ ‘[’ ‘]’

⟨*array access*⟩ ::= ⟨*expression name*⟩ ‘[’ ⟨*expression*⟩ ‘]’
  | ⟨*primary no new array*⟩ ‘[’ ⟨*expression*⟩ ‘]’

Tokens

⟨*expression name*⟩ ::= ⟨*identifier*⟩ | ⟨*ambiguous name*⟩ . ⟨*identifier*⟩

⟨*method name*⟩ ::= ⟨*identifier*⟩ | ⟨*ambiguous name*⟩ . ⟨*identifier*⟩

⟨*ambiguous name*⟩ ::= ⟨*identifier*⟩ | ⟨*ambiguous name*⟩ . ⟨*identifier*⟩

⟨*literal*⟩ ::= ⟨*integer literal*⟩ | ⟨*floating-point literal*⟩ | ⟨*boolean literal*⟩ | ⟨*character literal*⟩
| ⟨*string literal*⟩ | ⟨*null literal*⟩

⟨*integer literal*⟩ ::= ⟨*decimal integer literal*⟩ | ⟨*hex integer literal*⟩ | ⟨*octal integer literal*⟩

⟨*decimal integer literal*⟩ ::= ⟨*decimal numeral*⟩ ⟨*integer type suffix*⟩?

⟨*hex integer literal*⟩ ::= ⟨*hex numeral*⟩ ⟨*integer type suffix*⟩?

⟨*octal integer literal*⟩ ::= ⟨*octal numeral*⟩ ⟨*integer type suffix*⟩?

⟨*integer type suffix*⟩ ::= ‘`l`’ | ‘`L`’

⟨*decimal numeral*⟩ ::= 0 | ⟨*non zero digit*⟩ ⟨*digits*⟩?

⟨*digits*⟩ ::= ⟨*digit*⟩ | ⟨*digits*⟩ ⟨*digit*⟩

⟨*digit*⟩ ::= 0 | ⟨*non zero digit*⟩

⟨*non zero digit*⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*hex numeral*⟩ ::= 0 x ⟨*hex digit*⟩ | 0 X ⟨*hex digit*⟩ | ⟨*hex numeral*⟩ ⟨*hex digit*⟩

⟨*hex digit*⟩ :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

⟨*octal numeral*⟩ ::= 0 ⟨*octal digit*⟩ | ⟨*octal numeral*⟩ ⟨*octal digit*⟩

⟨*octal digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨*floating-point literal*⟩ ::= ⟨*digits*⟩ . ⟨*digits*⟩? ⟨*exponent part*⟩? ⟨*float type suffix*⟩?

⟨*digits*⟩ ::= ⟨*exponent part*⟩? ⟨*float type suffix*⟩?

⟨*exponent part*⟩ ::= ⟨*exponent indicator*⟩ ⟨*signed integer*⟩

⟨*exponent indicator*⟩ ::= e | E

⟨*signed integer*⟩ ::= ⟨*sign*⟩? ⟨*digits*⟩

⟨*sign*⟩ ::= + | -

⟨*float type suffix*⟩ ::= f | F | d | D

⟨*boolean literal*⟩ ::= ‘`true`’ | ‘`false`’

⟨*character literal*⟩ ::= ' ⟨*single character*⟩ ' | ' ⟨*escape sequence*⟩ '

⟨*single character*⟩ ::= ⟨*input character*⟩ except ' and \

⟨*string literal*⟩ ::= " ⟨*string characters*⟩? "

⟨*string characters*⟩ ::= ⟨*string character*⟩ | ⟨*string characters*⟩ ⟨*string character*⟩

⟨*string character*⟩ ::= ⟨*input character*⟩ except ' and \ | ⟨*escape character*⟩

⟨*null literal*⟩ ::= 'null'

⟨*keyword*⟩ ::= 'boolean' | 'char' | 'const' | 'double' | 'else' | 'final'
| 'float' | 'if' | 'int' | 'long' | 'new' | 'private' | 'protected'
| 'public' | 'return' | 'short' | 'static' | 'this' | 'void' | 'while'

A.2 Device Performance Case Study

```
1  public class DevPerf {
2
3    static int devicePerformanceClass = -1;
4    static int PERFORMANCE_CLASS_LOW=0;
5    static int PERFORMANCE_CLASS_AVERAGE=1;
6    static int PERFORMANCE_CLASS_HIGH=2;
7    static int PERFORMANCE_CLASS_VERY_HIGH=3;
8    static int performance_mode = 1, default_mode = 2, power_mode =
      ↪  3, high_power_mode = 4, mode;
9
10   public static int getDevicePerfomanceClass(int androidVersion,
      ↪  int cpuCount,
11   int memoryClass, int maxCpuFreq) {
12     if (devicePerformanceClass == -1) {
13        if (androidVersion < 21 || cpuCount <= 2 || memoryClass <=
           ↪  100 || cpuCount <= 4 && maxCpuFreq != -1 && maxCpuFreq
           ↪  <= 1250 || cpuCount <= 4 && maxCpuFreq <= 1600 &&
           ↪  memoryClass <= 128 && androidVersion <= 21 || cpuCount
           ↪  <= 4 && maxCpuFreq <= 1300 && memoryClass <= 128 &&
           ↪  androidVersion <= 24) {
```

```
14                  devicePerformanceClass = PERFORMANCE_CLASS_LOW;
15                  mode = performance_mode;
16                  Animations(mode); //@energy=28
17             } else {
18                  if (cpuCount < 8 || memoryClass <= 160 || maxCpuFreq
                     ↪  != -1 && maxCpuFreq <= 1650 || maxCpuFreq == -1 &&
                     ↪  cpuCount == 8 && androidVersion <= 23) {
19                       devicePerformanceClass =
                          ↪  PERFORMANCE_CLASS_AVERAGE;
20                       mode = default_mode;
21                       Animations(mode); //@energy=34
22                  } else {
23                       if (cpuCount < 8 || memoryClass <= 200 ||
                          ↪  maxCpuFreq != -1 && maxCpuFreq <= 1950 ||
                          ↪  maxCpuFreq == -1 && cpuCount == 8 &&
                          ↪  androidVersion <= 25){
24                            devicePerformanceClass =
                               ↪  PERFORMANCE_CLASS_HIGH;
25                            mode = power_mode;
26                            Animations(mode); //@energy=40
27   } else {
28                            devicePerformanceClass =
                               ↪  PERFORMANCE_CLASS_VERY_HIGH;
29                            mode = high_power_mode;
30                            Animations(mode); //@energy=48
31                       }
32                  }
33            }
34            }
35         return devicePerformanceClass;
36      }
37   }
```

Fig. A2.1: Java method `DevPerf` from the Telegram android application

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6  const double p4;
7
8  const int end_state = 17;
9
10 module DevPerf
11    s : [0..end_state] init 0;
12
13    [] s=0   ->  p1:(s'=1) + (1-p1):(s'=16);   //line:12
14    [] s=1   ->  p2:(s'=2) + (1-p2):(s'=5);    //line:15
15    [] s=2   ->  1:(s'=3);                     //line:16
16    [] s=3   ->  1:(s'=4);                     //line:17
17    [] s=4   ->  1:(s'=16);                    //line:18
18    [] s=5   ->  p3:(s'=6) + (1-p3):(s'=9);    //line:20
19    [] s=6   ->  1:(s'=7);                     //line:21
20    [] s=7   ->  1:(s'=8);                     //line:22
21    [] s=8   ->  1:(s'=16);                    //line:23
22    [] s=9   ->  p4:(s'=10) + (1-p4):(s'=13); //line:25
23    [] s=10  ->  1:(s'=11);                    //line:26
24    [] s=11  ->  1:(s'=12);                    //line:27
25    [] s=12  ->  1:(s'=16);                    //line:28
26    [] s=13  ->  1:(s'=14);                    //line:30
27    [] s=14  ->  1:(s'=15);                    //line:31
28    [] s=15  ->  1:(s'=16);                    //line:32
29    [] s=16  ->  1:(s'=end_state);             //line:38
30    [] s=17  ->  true;
31 endmodule
32
33 rewards "energy"
34    s=4  : 28;
35    s=8  : 34;
36    s=12 : 40;
37    s=15 : 48;
38 endrewards
```

Fig. A2.2: PRISM model synthesised for the `DevPerf` Java method

A.3 Fast Sine Transformer Case Study

```
1  public class FastSineTransformer {
2
3    public static double[] fst(double[] f) throws
      ↪ MathIllegalArgumentException {
4
5      double[] transformed = new double[f.length];
6      if (!ArithmeticUtils.isPowerOfTwo(f.length)) {
7          throw new MathIllegalArgumentException(
            ↪ LocalizedFormats.NOT_POWER_OF_TWO_CONSIDER_PADDING,
            ↪ Integer.valueOf(f.length)); //@cost=5
8      }
9
10     if (f[0] != 0.0) {
11        throw new MathIllegalArgumentException(
12        LocalizedFormats.FIRST_ELEMENT_NOT_ZERO,
           ↪ Double.valueOf(f[0])); //@cost=5
13     }
14
15     int n = f.length;
16     if (n == 1) {
17        transformed[0] = 0.0;
18        return transformed;
19     }
20
21     double[] x = new double[n];
22     x[0] = 0.0;
23     x[n >> 1] = 2.0 * f[n >> 1];
24     int i=1;
25     while (i < (n >> 1)) {
26        double a = FastMath.sin(i * FastMath.PI / n) * (f[i] + f[n
           ↪ - i]); //@time=1.5
27        double b = 0.5 * (f[i] - f[n - i]);
28        x[i]     = a + b;
29        x[n - i] = a - b;
30        i++;
31     }
32     FastFourierTransformer transformer;
33     transformer = new
        ↪ FastFourierTransformer(DftNormalization.STANDARD);
```

```
34        Complex[] y = transformer.transform(x,
           ↪  TransformType.FORWARD);
35
36        transformed[0] = 0.0;
37        transformed[1] = 0.5 * y[0].getReal();
38        int j = 1;
39        while (j < (n >> 1)) {
40            transformed[2 * j]     = -y[j].getImaginary();
41            transformed[2 * j + 1] = y[j].getReal() + transformed[2 *
               ↪  j - 1];
42            j++;
43        }
44
45        return transformed;
46        }
47    }
```

Fig. A3.1: Java method `fst` from the Apache Commons Math library

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6  const double p4;
7  const double p5;
8
9  const int end_state=30;
10
11 module FastSineTransformer
12     s : [0..end_state] init 0;
13
14     [] s=0   ->  1:(s'=1);                   //line:5
15     [] s=1   ->  p1:(s'=2) + (1-p1):(s'=3);  //line:6
16     [] s=2   ->  1:(s'=end_state);           //line:7
17     [] s=3   ->  p2:(s'=4) + (1-p2):(s'=5);  //line:10
18     [] s=4   ->  1:(s'=end_state);           //line:11
19     [] s=5   ->  1:(s'=6);                   //line:15
20     [] s=6   ->  p3:(s'=7) + (1-p3):(s'=9);  //line:16
21     [] s=7   ->  1:(s'=8);                   //line:17
22     [] s=8   ->  1:(s'=end_state);           //line:18
```

```
23      [] s=9    ->  1:(s'=10);                      //line:21
24      [] s=10   ->  1:(s'=11);                      //line:22
25      [] s=11   ->  1:(s'=12);                      //line:23
26      [] s=12   ->  1:(s'=13);                      //line:24
27      [] s=13   ->  p4:(s'=14)  +  (1-p4):(s'=19);  //line:25
28      [] s=14   ->  1:(s'=15);                      //line:26
29      [] s=15   ->  1:(s'=16);                      //line:27
30      [] s=16   ->  1:(s'=17);                      //line:28
31      [] s=17   ->  1:(s'=18);                      //line:29
32      [] s=18   ->  1:(s'=13);                      //line:30
33      [] s=19   ->  1:(s'=20);                      //line:32
34      [] s=20   ->  1:(s'=21);                      //line:33
35      [] s=21   ->  1:(s'=22);                      //line:34
36      [] s=22   ->  1:(s'=23);                      //line:36
37      [] s=23   ->  1:(s'=24);                      //line:37
38      [] s=24   ->  1:(s'=25);                      //line:38
39      [] s=25   ->  p5:(s'=26)  +  (1-p5):(s'=29);  //line:39
40      [] s=26   ->  1:(s'=27);                      //line:40
41      [] s=27   ->  1:(s'=28);                      //line:41
42      [] s=28   ->  1:(s'=25);                      //line:42
43      [] s=29   ->  1:(s'=end_state);               //line:45
44      [] s=30   ->  true;
45  endmodule
46
47  rewards "time"
48      s=14 : 2.6;
49  endrewards
50
51  rewards "cost"
52      s=2 : 5;
53      s=4 : 5;
54  endrewards
```

Fig. A3.2: PRISM model synthesised for the `fst` Java method

A.4 Knapsack Algorithm Case Study

```
1  public class Knapsack {
2
3     public int knapsackDP(int[] w, int[] v, int n, int W) {
4        if (n <= 0 || W <= 0) {
5           return 0;
6        }
7
8        int[][] m = new int[n + 1][W + 1];
9
10       int j = 0;
11       while (j <= W) {
12          m[0][j] = 0;
13          j++;
14       }
15
16       int i = 1;
17       while (i <= n) {
18          int k = 1;
19             while (k <= W) {
20                if (w[i - 1] > k) {
21                   m[i][k] = m[i - 1][k];
22                } else {
23                   m[i][k] = Math.max(m[i - 1][k], m[i - 1][k -
                      ↪ w[i - 1]] + v[i - 1]); //@time=2
24                   display(); // @energy:67
25                }
26                k++;
27             }
28             i++;
29          }
30         return m[n][W];
31    }
32 }
```

Fig. A4.1: Java implementation of the `knapsack` algorithm

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6  const double p4;
7  const double p5;
8
9  const int end_state=18;
10
11 module Knapsack
12     s : [0..end_state] init 0;
13
14     [] s=0   -> p1:(s'=1) + (1-p1):(s'=2);    //line:4
15     [] s=1   -> 1:(s'=end_state);             //line:5
16     [] s=2   -> 1:(s'=3);                     //line:8
17     [] s=3   -> 1:(s'=4);                     //line:10
18     [] s=4   -> p2:(s'=5) + (1-p2):(s'=7);    //line:11
19     [] s=5   -> 1:(s'=6);                     //line:12
20     [] s=6   -> 1:(s'=4);                     //line:13
21     [] s=7   -> 1:(s'=8);                     //line:16
22     [] s=8   -> p3:(s'=9) + (1-p3):(s'=17);   //line:17
23     [] s=9   -> 1:(s'=10);                    //line:18
24     [] s=10  -> p4:(s'=11) + (1-p4):(s'=16);  //line:19
25     [] s=11  -> p5:(s'=12) + (1-p5):(s'=13);  //line:20
26     [] s=12  -> 1:(s'=15);                    //line:21
27     [] s=13  -> 1:(s'=14);                    //line:23
28     [] s=14  -> 1:(s'=15);                    //line:24
29     [] s=15  -> 1:(s'=10);                    //line:26
30     [] s=16  -> 1:(s'=8);                     //line:28
31     [] s=17  -> 1:(s'=end_state);             //line:30
32     [] s=18  -> true;
33 endmodule
34
35 rewards "time"
36     s=13 : 2.0;
37 endrewards
38
39 rewards "energy"
40     s=14 : 67;
41 endrewards
```

Fig. A4.2: PRISM model synthesised for the `knapsack` Java implementation

Appendix B

Chapter 4 - Supplementary Material

B.1 Parametric DTMC Models

The parametric DTMC models listed below are examples of the models used for the extraction of the success probabilities for each of the external services, namely MW, FA and Order, and their response times. The results of the verification process were then supplied to the parametric CTMC model during the model analysis step (Section 4.2.3) of the antipattern-based analysis approach in Chapter 4. The above mentioned properties expressed in PCTL are:

(P_1) $P = ?[F \text{“}success\text{”}]$

(P_2) $R\{\text{“}response_time\text{”}\} = ?[F \text{“}done\text{”}]$

where “*success*” refers to the state that a service instance has successfully been invoked, and “*done*” refers to the final state of the model, which is reached after either failed or successful service invocation attempts.

In the models below, MW service instances (Figure B1.1) are invoked using with the parallel (PAR) pattern, i.e., all services instances are invoked simultaneously and the first successful invoked instance is selected. FA service instances (Figure B1.2) are invoked using the sequential (SEQ) pattern, i.e., the first invoked service instance must fail in order for the second to be invoked, etc. Order service instance (Figure B1.3) are invoked using the probabilistic (PROB) pattern, i.e., the services are selected based on a discrete probability distribution.

The patterns of invoking the services' instances are not tied to a specific service, and they can be used in any combination of service-pattern deemed suitable. For example, in our experimentation in Section 4.4.2 refactoring actions indicated the need for changing

the service pattern in some external services to satisfy the system's requirements. Note that the Order operation is not idempotent, so the parallel option cannot be used for it.

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6  const double rt1;
7  const double rt2;
8  const double rt3;
9  const double timeout;
10
11 module MARKET_WATCH
12   s : [1..9] init 1;
13
14   [] s=1 -> p1:(s'=4) + (1-p1):(s'=2); // MW_1
15   [] s=2 -> p2:(s'=5) + (1-p2):(s'=3); // MW_2
16   [] s=3 -> p3:(s'=6) + (1-p3):(s'=7); // MW_3
17
18   [] s=4 -> (s'=8); // MW_1 success
19   [] s=5 -> (s'=8); // MW_2 success
20   [] s=6 -> (s'=8); // MW_3 success
21
22   [] s=7 -> (s'=9); // failure
23   [] s=8 -> (s'=9); // success
24   [] s=9 -> (s'=9); // done
25 endmodule
26
27 rewards "rt"
28   s=4  : rt1;
29   s=5  : rt2;
30   s=6  : rt3;
31   s=7 : timeout;
32 endrewards
```

Fig. B1.1: DTMC model for MW synthesised using the PAR pattern

```
1  dtmc
2
3  const double p1;
4  const double p2;
5  const double p3;
6  const double rt1;
7  const double rt2;
8  const double rt3;
9  const double delay;
10
11 module FUNDAMENTAL_ANALYSIS
12   s : [1..12] init 1;
13
14   [] s=1 -> p1:(s'=4) + (1-p1):(s'=5); // FA_1
15   [] s=2 -> p2:(s'=6) + (1-p2):(s'=7); // FA_2
16   [] s=3 -> p3:(s'=8) + (1-p3):(s'=9); // FA_3
17
18   [] s=4 -> (s'=10); // FA_1 success
19   [] s=5 -> (s'=2); // failure, go to next impl
20
21   [] s=6 -> (s'=10); // FA_2 success
22   [] s=7 -> (s'=3); // failure, go to next impl
23
24   [] s=8 -> (s'=10); // FA_3 success
25   [] s=9 -> (s'=11); // failure, go to next impl
26
27   [] s=10 -> (s'=12); // success
28   [] s=11 -> (s'=12); // failure
29   [] s=12 -> (s'=12); // done
30 endmodule
31
32 rewards "rt"
33   s=4 : rt1;
34   s=6 : rt2;
35   s=8 : rt3;
36   s=5|s=7|s=9 : delay;
37 endrewards
```

Fig. B1.2: DTMC model for FA synthesised using the SEQ pattern

```
1  dtmc
2
3  const double x1;
4  const double x2;
5  const double x3=1-x1-x2;
6
7  const double p1;
8  const double p2;
9  const double p3;
10
11 const double rt1;
12 const double rt2;
13 const double rt3;
14 const double timeout;
15
16 module ORDER
17   s : [1..10] init 1;
18
19   [] s=1 -> x1:(s'=2) + x2:(s'=3) + x3:(s'=4); // select one of
       the N=3 implementations based on a discrete probability
       distribution
20
21   [] s=2 -> p1:(s'=5) + (1-p1):(s'=8); // O1
22   [] s=3 -> p2:(s'=6) + (1-p2):(s'=8); // O2
23   [] s=4 -> p3:(s'=7) + (1-p3):(s'=8); // O3
24
25   [] s=5 -> (s'=9);  // success O1
26   [] s=6 -> (s'=9);  // success O2
27   [] s=7 -> (s'=9);  // success O3
28
29   [] s=8 -> (s'=10);  // timeout failure
30   [] s=9 -> (s'=10); // success
31   [] s=10 -> (s'=10); // done
32 endmodule
33
34 rewards "rt"
35   s=5  : rt1;
36   s=6  : rt2;
37   s=7  : rt3;
38   s=8  : timeout;
39 endrewards
```

Fig. B1.3: DTMC model for Order synthesised using the PROB pattern

B.2 Parametric CTMC Model

The CTMC model depicted in this section represents the FX system with one additional thread and instance of the TA component. It was used during the Evaluation (Section 4.4) of the antipattern-based approach in Chapter 4 as part of the second refactoring action (R_2), which introduced the addition of a second thread in the system.

```
1  ctmc
2
3  const double pObjNotMet;
4  const double pObjSatisfied;
5  const double pObjNotMetHighVar = 1-pObjSatisfied-pObjNotMet;
6  const double pExpertMode;
7  const double pFAMode = 1-pExpertMode;
8  const double pPerformTransaction;
9  const double MWsucc;
10 const double FASucc;
11 const double OrderSucc;
12 const double MWrate;
13 const double FARate;
14 const double OrderRate;
15 const double reqRate;
16 const int nthreads;
17 const int MAX_QUEUE_SIZE;
18 const double ta1Rate;
19 const double ta2Rate;
20 const double alarmRate;
21 const double notifRate;
22 const double internalOpRate;
23
24 module RequestQueue
25   q : [0..MAX_QUEUE_SIZE] init 0;
26
27   [NewReq]     true -> reqRate : (q'=min(q+1,MAX_QUEUE_SIZE));
28   //thread 1 and 2 serving requests
29   [ServeReq1] q>0 -> internalOpRate:(q'=q-1);
30   [ServeReq2] q>0 -> internalOpRate:(q'=q-1);
31 endmodule
```

```
32 module Workflow1
33   s1 : [0..10] init 0;
34
35   [ServeReq1] s1=0 -> pExpertMode:(s1'=1) + pFAMode:(s1'=9);
36
37   [MW1] s1=1 -> MWsucc*MWrate:(s1'=2)+(1-MWsucc)*MWrate:(s1'=10);
38
39   [TA1Invoke1] s1=2 -> 1:(s1'=3); //thread 1 invokes TA1 instance
40   [TA1Invoke1] s1=2 -> 1:(s1'=4); //thread 1 invokes TA2 instance
41   [TA1Exec1]   s1=3 -> pObjSatisfied:(s1'=7) + pObjNotMet:(s1'=1)
       + pObjNotMetHighVar:(s1'=5); // wait for TA1 to complete
42   [TA1Exec1]   s1=4 -> pObjSatisfied:(s1'=7) + pObjNotMet:(s1'=1)
       + pObjNotMetHighVar:(s1'=5); // wait for TA2 to complete
43
44   [Alarm1] s1=5 -> alarmRate:(s1'=6);
45
46   [] s1=6 -> internalOpRate:(s1'=0);
47
48   [Order1] s1=7 -> OrderSucc*OrderRate:(s1'=8) + (1-OrderSucc)*
      OrderRate:(s1'=10);
49
50   [Notif1] s1=8 -> notifRate:(s1'=6);
51
52   [FA1] s1=9 -> FASucc*FARate*pPerformTransaction:(s1'=7) +
      FASucc*FARate*(1-pPerformTransaction):(s1'=6) + (1-FASucc)*
      FARate:(s1'=10);
53
54   [] s1=10 -> internalOpRate:(s1'=0);
55 endmodule
56
57 // Other workflow threads
58 module Workflow2 = Workflow1[s1=s2, ServeReq1=ServeReq2, MW1=MW2,
       TA1Invoke1=TA1Invoke2, TA1Invoke1=TA1Invoke2, TA1Exec1=TA1
      Exec2, TA1Exec1=TA1Exec2, Alarm1=Alarm2, Order1=Order2, Notif1
      =Notif2, FA1=FA2] endmodule
59
60 // Internal component Technical Analysis, instance 1
61 module TA1
62   t1 : [0..2] init 0;
63   [TA1Invoke1] t1=0 -> internalOpRate:(t1'=1);
64   [TA1Invoke2] t1=0 -> internalOpRate:(t1'=2);
65   [TA1Exec1]   t1=1 -> ta1Rate:(t1'=0);
66   [TA1Exec2]   t1=2 -> ta1Rate:(t1'=0);
67 endmodule
```

```
68  // Internal component Technical Analysis, instance 2
69  module TA1 = TA1 [t1=t2, TA1Invoke1=TA1Invoke1, TA1Invoke2=TA1
        Invoke2, TA1Exec1=TA1Exec1, TA1Exec2=TA1Exec2, ta1Rate=ta2Rate
        ] endmodule
70
71  rewards "droppedRequests"
72    [NewReq] q=MAX_QUEUE_SIZE:1;
73  endrewards
74
75  rewards "numOfReqsHandled"
76    [NewReq] q<MAX_QUEUE_SIZE:1;
77  endrewards
78
79  rewards "MWcount"
80    [MW1] true : 1;
81    [MW2] true : 1;
82  endrewards
83
84  rewards "Notifcount"
85    [Notif1] true : 1;
86    [Notif2] true : 1;
87  endrewards
88
89  rewards "Alarmcount"
90    [Alarm1] true : 1;
91    [Alarm2] true : 1;
92  endrewards
93
94  rewards "FAcount"
95    [FA1] true : 1;
96    [FA2] true : 1;
97  endrewards
98
99  rewards "Ordercount"
100   [Order1] s1=7 : 1;
101   [Order2] s2=7 : 1;
102 endrewards
103
104 rewards "served"
105   [ServeReq1] true : 1;
106   [ServeReq2] true : 1;
107 endrewards
108
109 rewards "extFails"
110   s1=10 : 1;
111   s2=10 : 1;
112 endrewards
113
114 rewards "processingTime"
115   s1>0 : 1;
116   s2>0 : 1;
117 endrewards
118
119 rewards "idle"
120   s1=0 : 1;
121   s2=0 : 1;
122 endrewards
123
124 rewards "qLen"
125   true : q;
126 endrewards
127
128 rewards "servedTA"
129   [TA1Invoke1] true : 1;
130   [TA1Invoke2] true : 1;
131   [TA1Invoke1] true : 1;
132   [TA1Invoke2] true : 1;
133 endrewards
134
135 rewards "taTime"
136   s1=2 : 1;
137   s1=3 : 1;
138   s2=2 : 1;
139   s2=3 : 1;
140 endrewards
```

Fig. B2.1: CTMC model representation of the FX system consisted of two threads and two instances of TA component

B.3 Graphs for the CTMC Model's Properties

This section contains generated graphs for selected properties of the CTMC model, verified using the PRISM model checker. These graphs are generated based on the initial configuration of the system, i.e., varying the probability of executing the workflow in expert mode (p_{EM}) and the probability of successfully performing a transaction (p_{PT}) between 0.01 and 1 with a step of 0.1, and the probabilities of satisfying or not the objectives are $pObjectivesSatisfied = 0.21$ and $pObjectivesNotMet = 0.78$, respectively.

Figure B3.1 shows the number of invocations for the Order external service in $T = 10$ time-steps. The maximum number of Order's invocations is reached when (a) the probability of successfully performing a transaction (p_{PT}) also reaches its maximum value (0.91), and (b) the probability of operating in expert mode is the lowest, i.e., the system operates in fundamental analysis mode. This is due to the low probability of satisfying the objectives set in expert mode ($pObjectivesSatisfied = 0.21$), and for high values of p_{EM} we see that p_{PT} does not affect the number of Order's invocations as expected.

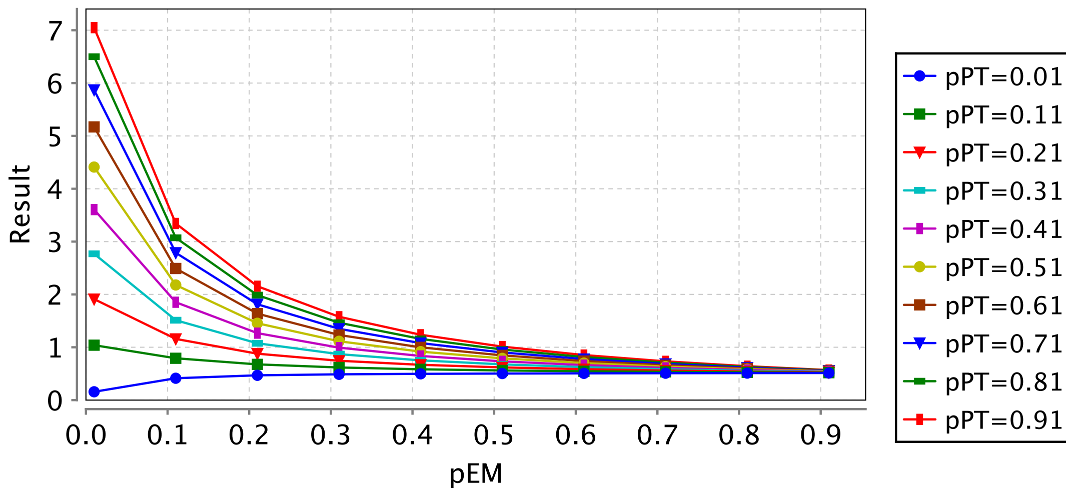


Fig. B3.1: Number of Order service invocations per time unit

Figure B3.2 depicts the probability of executing the FA $\rightarrow$ Order $\rightarrow$ Notification path, while varying the system's parameters. The lower p_{EM} is, the higher is the probability of executing the fundamental analysis branch, and thus, the probability of invoking the listed components in the mentioned order. Also, p_{PT} affects the path probability as it dictates the occurrence of transactions, leading to the invocation of the Order external service, and following, the Notification internal component.

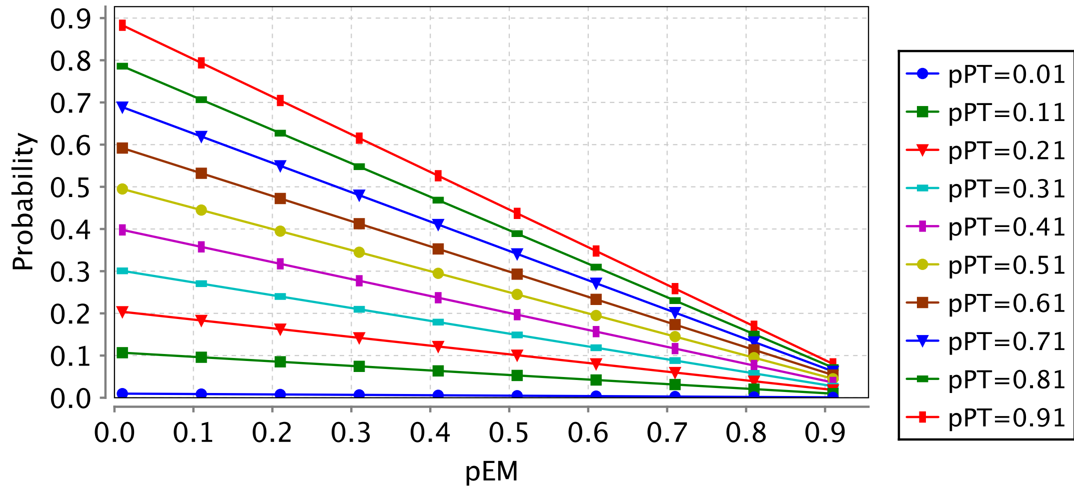


Fig. B3.2: The path probability of executing the FA and Order external services, and the Notification internal component

Figure B3.3 shows the response time of the TA component while varying the operational profile of the system. As expected, when the probability of executing the workflow in expert mode increases, TA's response time also increases as more requests are going through that branch of the system. Additionally, we observe that the probability of performing a transaction (p_{PT}) has a minimal effect on TA's response time, and when the value of p_{EM} increases all various p_{PT} cases merge.

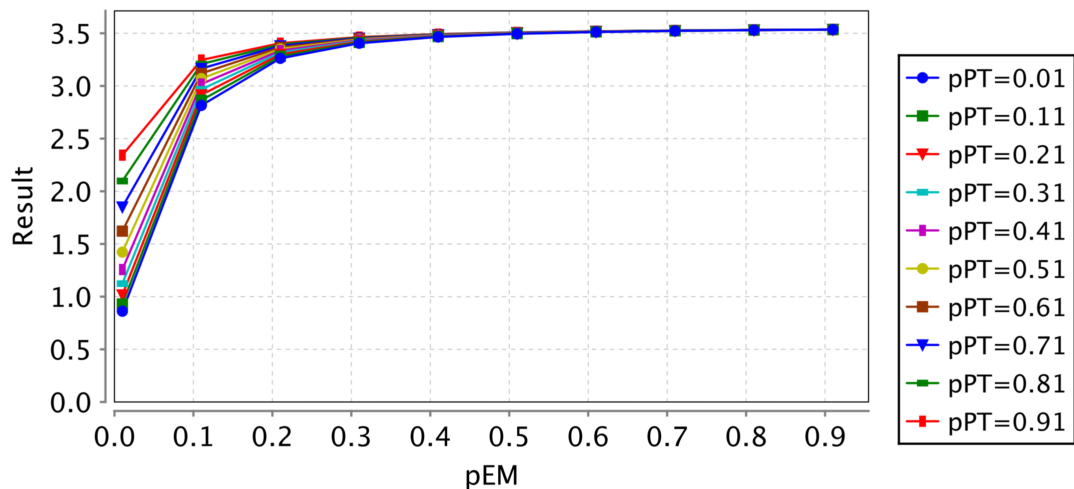


Fig. B3.3: TA internal component's response time

References

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2012.

[2] A. Aleti, C. Trubiani, A. van Hoorn, and P. Jamshidi, "An efficient method for uncertainty propagation in robust software performance estimation," *Journal of Systems and Software*, vol. 138, pp. 222–235, 2018.

[3] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *ASE'18*, 2018, pp. 464–474.

[4] P. Arafa, G. M. Tchamgoue, H. Kashif, and S. Fischmeister, "QDIME: QoS-aware dynamic binary instrumentation," in *MASCOTS*, 2017, pp. 132–142.

[5] D. Arcelli, "Exploiting queuing networks to model and assess the performance of self-adaptive software systems: a survey," *Procedia Computer Science*, vol. 170, pp. 498–505, 2020.

[6] D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," in *QoSA'12*, 2012, pp. 33–42.

[7] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Verifying continuous time Markov chains," in *Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds. Springer Berlin Heidelberg, 1996, pp. 269–276.

[8] A. Aziz, V. Singhal, and F. Balarin, "It usually works: The temporal logic of stochastic systems," in *Proceedings of the 7th International Conference on Computer Aided Verification*. Springer-Verlag, 1995, p. 155–165.

[9] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[10] C. Baier, J.-P. Katoen, and H. Hermanns, "Approximative symbolic model checking of continuous-time Markov chains," in *CONCUR'99 Concurrency Theory*, 1999, pp. 146–161.

[11] T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," *TOPLAS*, vol. 16, no. 4, pp. 1319–1360, 1994.

[12] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based Performance Prediction in Software Development: A Survey," *Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[13] L. Baresi and M. Pezzè, "An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, vol. 148, pp. 89–111, 2006.

[14] B. Bartels, U. Ermel, P. Sandborn, and M. G. Pecht, *Strategies to the prediction, mitigation and management of product obsolescence*. John Wiley & Sons, 2012, vol. 87.

[15] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[16] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[17] K. H. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000*, 2000, pp. 73–87.

[18] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and analysis of software systems specified with UML," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 2:1–2:48, 2012.

[19] A. Bertolino, "Software testing research and practice," in *Abstract State Machines 2003*. Springer Berlin Heidelberg, 2003, pp. 1–21.

[20] ——, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*, 2007, pp. 85–103.

[21] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[22] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu, "Model-counting approaches for nonlinear numerical constraints," in *NASA Formal Methods*, 2017, pp. 131–138.

[23] P. Bratley, B. L. Fox, and L. E. Schrage, *A Guide to Simulation*. Springer-Verlag, 1987.

[24] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, 1998.

[25] W. J. Brown, H. W. McCormick, and S. H. Thomas, *AntiPatterns and Patterns in Software Configuration Management*. John Wiley and Sons, Inc., 1999.

[26] A. Busch, D. Fuchss, and A. Koziolek, "Peropteryx: Automated improvement of software architectures," in *ICSA-C'19*, 2019, pp. 162–165.

[27] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, and G. Tamburrelli, "Formal verification with confidence intervals to establish quality of service properties of software systems," *IEEE Transactions on Reliability*, vol. 65, pp. 107–125, 2016.

[28] R. Calinescu, V. Cortellessa, I. Stefanakos, and C. Trubiani, “Analysis and refactoring of software systems using performance antipattern profiles,” in *23rd International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2020, pp. 357–377.

[29] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola, “Self-adaptive software needs quantitative verification at runtime,” *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[30] R. Calinescu, K. Johnson, and C. Paterson, “FACT: A probabilistic model checker for formal verification with confidence intervals,” in *TACAS*, 2016, pp. 540–546.

[31] R. Calinescu, M. C. Jr., S. Gerasimou, M. Kwiatkowska, and N. Paoletti, “Efficient synthesis of robust models for stochastic systems,” *Journal of Systems and Software*, vol. 143, pp. 140–158, 2018.

[32] R. Calinescu and S. Kikuchi, “Formal methods @ runtime,” in *Monterey Workshop*. Springer, 2010, pp. 122–135.

[33] R. Calinescu and M. Kwiatkowska, “CADS*: Computer-aided development of self-* systems,” in *FASE’09*. Springer, 2009, pp. 421–424.

[34] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, “Engineering trustworthy self-adaptive software with dynamic assurance cases,” *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1039–1069, 2018.

[35] J. Camara, D. Garlan, and B. Schmerl, “Synthesis and quantitative verification of tradeoff spaces for families of software systems,” in *Software Architecture*. Springer International Publishing, 2017, pp. 3–21.

[36] M. Caporuscio, M. D’Angelo, V. Grassi, and R. Mirandola, “Decentralized architecture for energy-aware service assembly,” in *Software Architecture*. Springer International Publishing, 2020, pp. 57–72.

[37] G. Casella and R. L. Berger, *Statistical inference*. Cengage Learning, 2021.

[38] G. Chang, C. Tan, G. Li, and C. Zhu, “Developing mobile applications on the android platform,” in *Workshop of Mobile Multmedia Processing*. Springer, 2008, pp. 264–286.

[39] X. Chen, Z. Zheng, Q. Yu, and M. R. Lyu, “Web service recommendation via exploiting location and qos information,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 7, pp. 1913–1924, 2014.

[40] E. M. Clarke, “The birth of model checking,” in *Lecture Notes in Computer Science*, vol. 5000, 2008, pp. 1–26.

[41] E. M. Clarke, E. A. Emerson, and J. Sifakis, “Model checking: Algorithmic verification and debugging,” *Communications of the ACM*, vol. 52, no. 11, p. 74–84, 2009.

[42] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[43] E. M. Clarke and F. Lerda, "Model checking: Software and beyond," *J. Ucs*, vol. 13, no. 5, pp. 639–649, 2007.

[44] R. Coleman, "What is a stochastic process?" in *Stochastic Processes*. Springer Netherlands, 1974, pp. 1–5.

[45] K. Cooper, L. Dai, and Y. Deng, "Performance modeling and analysis of software architectures: An aspect-oriented UML based approach," *Science of Computer Programming*, vol. 57, no. 1, pp. 89–108, 2005.

[46] J. O. Coplien, "Software design patterns," in *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003, p. 1604–1606.

[47] V. Cortellessa, A. D. Marco, and P. Inverardi, "Non-functional modeling and validation in model-driven architecture," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, 2007, pp. 25–25.

[48] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into UML models to remove performance antipatterns," in *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, 2010, p. 9–16.

[49] V. Cortellessa, A. Di Marco, and P. Inverardi, "Integrating performance and reliability analysis in a non-functional MDA framework," in *Fundamental Approaches to Software Engineering*, 2007, pp. 57–71.

[50] V. Cortellessa, A. D. Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer, 2011.

[51] V. Cortellessa, A. D. Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *Software and Systems Modeling*, vol. 13, no. 1, pp. 391–432, 2014.

[52] V. Cortellessa, F. Marinelli, and P. Potena, "Automated selection of software components based on cost/reliability tradeoff," in *Software Architecture*, 2006, pp. 66–81.

[53] V. Cortellessa, R. Mirandola, and P. Potena, "Selecting optimal maintenance plans based on cost/reliability tradeoffs for software subject to structural and behavioral changes," in *14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 21–30.

[54] V. Cortellessa, D. D. Pompeo, V. Stoico, and M. Tucci, "On the impact of performance antipatterns in multi-objective software model refactoring optimization," *CoRR*, vol. abs/2107.06127, 2021. [Online]. Available: https://arxiv.org/abs/2107.06127

[55] V. Cortellessa, C. Trubiani, L. Mostarda, and N. Dulay, "An architectural framework for analyzing tradeoffs between software security and performance," in *Architecting Critical Systems*, 2010, pp. 1–18.

[56] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *FSE*, 2012, pp. 1–11.

[57] P. Cousot, “Methods and logics for proving programs,” in *Formal Models and Semantics*, ser. Handbook of Theoretical Computer Science. Elsevier, 1990, pp. 841–993.

[58] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu *et al.*, “Software engineering for self-adaptive systems: A second research roadmap,” in *Software Engineering for Self-Adaptive Systems*. Springer Berlin Heidelberg, 2013, pp. 1–32.

[59] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Ábrahám, “Prophesy: A probabilistic parameter synthesis tool,” in *Computer Aided Verification*. Springer International Publishing, 2015, pp. 214–231.

[60] C. Dehnert, S. Junges, J. Katoen, and M. Volk, “A storm is coming: A modern probabilistic model checker,” in *CAV*, 2017, pp. 592–600.

[61] C. Dubslaff, A. Morozov, C. Baier, and K. Janschek, “Reduction methods on probabilistic control-flow programs for reliability analysis,” *CoRR*, vol. abs/2004.06637, 2020.

[62] M. B. Dwyer, A. Filieri, J. Geldenhuys, M. J. Gerrard, C. S. Pasareanu, and W. Visser, “Probabilistic program analysis,” in *Grand Timely Topics in Software Engineering - International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures*, 2015, pp. 1–25.

[63] E. A. Emerson, “The beginning of model checking: A personal perspective,” in *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Berlin Heidelberg, 2008, pp. 27–45.

[64] E. A. Emerson and J. Y. Halpern, "“Sometimes” and “not never” revisited: on branching versus linear time temporal logic," *Journal of the ACM*, vol. 33, pp. 151–178, 1986.

[65] B. Evans and D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*. O’Reilly Media, 2018.

[66] M. Famelis and M. Chechik, “Managing design-time uncertainty,” *Software and Systems Modeling*, vol. 18, no. 2, pp. 1249–1284, 2019.

[67] H. Fecher, M. Leucker, and V. Wolf, “Don’t know in probabilistic systems,” in *Model Checking Software*. Springer Berlin Heidelberg, 2006, pp. 71–88.

[68] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, “Reliability analysis of component-based systems with multiple failure modes,” in *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2010, pp. 1–20.

[69] A. Filieri, C. Pasareanu, and W. Visser, “Reliability analysis in Symbolic Pathfinder,” in *ICSE*, 2013, pp. 622–631.

[70] A. Filieri, C. S. Pasareanu, and G. Yang, “Quantification of software changes through probabilistic symbolic execution,” in *ASE*, 2015, pp. 703–708.

[71] J. M. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan, “Improving self-adaptation planning through software architecture-based stochastic modeling,” *Journal of Systems and Software*, vol. 115, pp. 42–60, 2016.

[72] E. Gamma, “Design patterns – ten years later,” in *Software Pioneers: Contributions to Software Engineering*. Springer Berlin Heidelberg, 2002, pp. 688–700.

[73] J. Geldenhuys, M. B. Dwyer, and W. Visser, “Probabilistic symbolic execution,” in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 166–176.

[74] S. Gerasimou, R. Calinescu, and G. Tamburrelli, “Synthesis of probabilistic models for quality-of-service software engineering,” *Autom. Softw. Eng.*, vol. 25, no. 4, pp. 785–831, 2018.

[75] S. Gerasimou, G. Tamburrelli, and R. Calinescu, “Search-based synthesis of probabilistic models for quality-of-service software engineering (t),” in *30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 319–330.

[76] I. Gerostathopoulos, T. Vogel, D. Weyns, and P. Lago, “How do we evaluate self-adaptive software systems?: A ten-year perspective of seams,” in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2021, pp. 59–70.

[77] S. Ghaith, M. Wang, P. Perry, Z. M. Jiang, P. O’Sullivan, and J. Murphy, “Anomaly detection in performance regression testing by transaction profile estimation,” *Softw. Test., Verif. Reliab.*, vol. 26, no. 1, pp. 4–39, 2016.

[78] C. Ghezzi and A. Molzam Sharifloo, “Dealing with non-functional requirements for adaptive systems via dynamic software product-lines,” in *Software Engineering for Self-Adaptive Systems*. Springer Berlin Heidelberg, 2013, pp. 191–213.

[79] C. Ghezzi and A. Molzam Sharifloo, “Model-based verification of quantitative non-functional properties for software product lines,” *Information and Software Technology*, vol. 55, no. 3, pp. 508–524, 2013.

[80] S. S. Gokhale and Michael Rung-Tsong Lyu, “A simulation approach to structure-based software reliability analysis,” *IEEE Transactions on Software Engineering*, vol. 31, no. 8, pp. 643–656, 2005.

[81] C. P. Gomes, A. Sabharwal, and B. Selman, “Model counting: A new strategy for obtaining good bounds,” in *AAAI*, vol. 10, 2006, pp. 1 597 538–1 597 548.

[82] ——, “Model counting,” in *Handbook of Satisfiability*. IOS press, 2021, pp. 993–1014.

[83] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, “Probabilistic programming,” in *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, 2014, pp. 167–181.

[84] V. Grassi, R. Mirandola, and A. Sabetta, "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach," *Journal of Systems and Software*, vol. 80, no. 4, pp. 528–558, 2007.

[85] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, pp. 512–535, 1994.

[86] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker storm," 2020.

[87] S. Holla and M. M. Katti, "Android based mobile application development and its security," *International Journal of Computer Trends and Technology*, pp. 486–490, 2012.

[88] C. Hsu and C. Huang, "An adaptive reliability analysis using path testing for complex component-based software systems," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 158–170, 2011.

[89] *IEEE Standard for System and Software Verification and Validation*, 2012, IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004).

[90] R. G. Ingalls, "Introduction to simulation," in *Proceedings of the 2011 Winter Simulation Conference (WSC)*, 2011, pp. 1374–1388.

[91] A. Itai and M. Rodeh, "Symmetry breaking in distributed networks," *Information and Computation*, vol. 88, no. 1, pp. 60–87, 1990.

[92] D. N. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, "How fast and fat is your probabilistic model checker? an experimental performance comparison," in *Hardware and Software: Verification and Testing*, 2008, pp. 69–85.

[93] S. Junges, E. Abraham, C. Hensel, N. Jansen, J.-P. Katoen, T. Quatmann, and M. Volk, "Parameter synthesis for Markov models," 2019. [Online]. Available: https://arxiv.org/abs/1903.07993

[94] S. Kamavaram and K. Goseva-Popstojanova, "Sensitivity of software usage to changes in the operational profile," in *Annual Workshop of NASA Goddard Software Engineering*, 2003, pp. 157–164.

[95] J.-P. Katoen, "The probabilistic model checking landscape," in *Symposium on Logic in Computer Science*, 2016, pp. 31–45.

[96] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The ins and outs of the probabilistic model checker mrmc," in *International Conference on the Quantitative Evaluation of Systems*, 2009, pp. 167–176.

[97] R. Kemmerer, "Integrating formal methods into the development process," *IEEE Software*, vol. 7, no. 5, pp. 37–50, 1990.

[98] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *NOMS'12*, 2012, pp. 1287–1294.

[99] R. H. Khan and P. E. Heegaard, "Translation from UML to Markov model: A performance modeling framework," in *Innovations in Computing Sciences and Software Engineering*. Springer Netherlands, 2010, pp. 365–371.

[100] H. Kleine Büning and T. Letterman, *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[101] R. L. Klevans and W. J. Stewart, "From queueing networks to Markov chains: The xmarca interface," *Performance Evaluation*, vol. 24, no. 1, pp. 23–45, 1995.

[102] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance evaluation*, vol. 67, pp. 634–658, 2010.

[103] N. Kumar, B. R. Childers, and M. L. Soffa, "Low overhead program monitoring and profiling," *SIGSOFT*, vol. 31, no. 1, pp. 28–34, 2005.

[104] M. Kwiatkowska, "Quantitative verification: Models, techniques and tools," in *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM Press, 2007, pp. 449–458.

[105] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation*, ser. LNCS (Tutorial Volume), vol. 4486, 2007, pp. 220–270.

[106] M. Kwiatkowska and D. Parker, "Advances in probabilistic model checking," in *Software Safety and Security - Tools for Analysis and Verification*, vol. 33, 2012, pp. 126–151.

[107] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *Computer Performance Evaluation: Modelling Techniques and Tools*, 2002, pp. 200–204.

[108] ——, "Advances and Challenges of Probabilistic Model Checking," in *48th Annual Allerton Conference on Communication, Control, and Computing*, 2010, pp. 1691–1698.

[109] ——, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification*, 2011, pp. 585–591.

[110] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2010, pp. 23–37.

[111] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[112] P. Lee, S. Verma, and I. Harris, "A Comparison of Error Detection between Simulation-based Validation and Model Checking," Center for Embedded Computer Systems, University of California, Tech. Rep., 2013.

[113] M. Luckcuck, A. Wellings, and A. Cavalcanti, “Safety-critical Java: level 2 in practice,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 6, p. e3951, 2017.

[114] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, “Modeling software architectures in the unified modeling language,” *ACM Transactions on Software Engineering and Methodology*, vol. 11, pp. 2–57, 2002.

[115] S. M. Mirsky, E. H. Groundwater, J. E. Hayes, and L. A. Miller, “Guidelines for the verification and validation of expert system software and conventional software: Survey and assessment of conventional software verification and validation methods,” Nuclear Regulatory Commission; Electric Power Research Institute, Tech. Rep., 1995.

[116] P. N. Misra, “Software reliability analysis,” *IBM Systems Journal*, vol. 22, no. 3, pp. 262–270, 1983.

[117] A. Miyazawa, A. Cavalcanti, and A. Wellings, “SCJ-Circus: Specification and refinement of Safety-Critical Java programs,” *Science of Computer Programming*, vol. 181, pp. 140–176, 2019.

[118] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, “Java programming for high-performance numerical computing,” *IBM Systems Journal*, vol. 39, no. 1, pp. 21–56, 2000.

[119] J. Musa, “Operational profiles in software-reliability engineering,” *IEEE Software*, vol. 10, no. 2, pp. 14–32, 1993.

[120] J. Musgrove, B. Cukic, and V. Cortellessa, “Proactive model-based performance analysis and security tradeoffs in a complex system,” in *15th International Symposium on High-Assurance Systems Engineering*, 2014, pp. 211–215.

[121] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[122] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, “Caramel: Detecting and fixing performance problems that have non-intrusive fixes,” in *ICSE’15*, 2015, pp. 902–912.

[123] G. Norman and D. Parker, “Quantitative verification: Formal guarantees for timeliness, reliability and performance,” London Mathematical Society and Smith Institute, Technical Report, 2014.

[124] J. R. Norris, *Markov Chains*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.

[125] A. Ouni, M. Kessentini, M. Ó. Cinnéide, H. A. Sahraoui, K. Deb, and K. Inoue, “MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells,” *Journal of Software: Evolution and Process*, vol. 29, no. 5, 2017.

[126] S. Özekici and R. Soyer, “Reliability of software with an operational profile,” *European Journal of Operational Research*, vol. 149, no. 2, pp. 459–474, 2003.

[127] T. Pasternak, "Using trade-off analysis to uncover links between functional and non-functional requirements in use-case analysis," in *Proceedings 2003 Symposium on Security and Privacy*, 2003, pp. 3–9.

[128] C. Paterson and R. Calinescu, "Observation-enhanced qos analysis of component-based systems," *IEEE Transactions on Software Engineering*, pp. 526–548, 2020.

[129] ——, "Accurate analysis of quality properties of software with observation-based Markov chain refinement," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 121–130.

[130] R. Pinciroli, C. U. Smith, and C. Trubiani, "Qn-based modeling and analysis of software performance antipatterns for cyber-physical systems," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2021, p. 93–104.

[131] M. Plauth, L. Feinbube, and A. Polze, "A performance survey of lightweight virtualization techniques," in *European Conference on Service-Oriented and Cloud Computing*, 2017, pp. 34–48.

[132] S. Ross, *Stochastic processes*, 2nd ed., ser. Series in probability and statistics. Wiley, 1996.

[133] S. Ruland, G. Kulcsár, E. Leblebici, S. Peldszus, and M. Lochau, "Controlling the attack surface of object-oriented refactorings," in *FASE'18*, 2018, pp. 38–55.

[134] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker, *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.), ser. CRM Monograph Series. American Mathematical Society, 2004, vol. 23.

[135] P. Ryan and C. T. Sennett, Eds., *Formal Methods in Systems Engineering*. Springer, 1993.

[136] M. Saadatmand, A. Cicchetti, and M. Sjödin, "Model-based trade-off analysis of non-functional requirements: An automated UML-based approach," *International Journal of Advanced Computer Science*, pp. 575–588, 2013.

[137] A. A. A. Saeed and S.-W. Lee, "Non-functional requirements trade-off in self-adaptive systems," *4th International Workshop on Requirements Engineering for Self-Adaptive, Collaborative, and Cyber Physical Systems*, pp. 9–15, 2018.

[138] M. D. Sanctis, C. Trubiani, V. Cortellessa, A. D. Marco, and M. Flamminj, "A model-driven approach to catch performance antipatterns in ADL specifications," *Information & Software Technology*, vol. 83, pp. 35–54, 2017.

[139] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin, "Profiling software for energy consumption," in *GreenCom*, 2012, pp. 515–522.

[140] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A Bayesian approach to reliability prediction and assessment of component based systems," in *Proceedings 12th International Symposium on Software Reliability Engineering*, 2001, pp. 12–21.

[141] B. Smaalders, "Performance anti-patterns: Want your apps to run faster? here's what not to do." *Queue*, vol. 4, p. 44–50, 2006.

[142] C. Smidts, C. Mutha, M. Rodríguez, and M. J. Gerber, "Software testing with an operational profile: OP definition," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 39, 2014.

[143] C. U. Smith, "Applying synthesis principles to create responsive software systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1394–1408, 1988.

[144] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Second International Workshop on Software and Performance*, 2000, pp. 127–136.

[145] ——, "New book - performance solutions: A practical guide to creating responsive, scalable software," in *27th International Computer Measurement Group Conference*. Computer Measurement Group, 2001, pp. 355–358.

[146] ——, "New software performance antipatterns: More ways to shoot yourself in the foot," in *28th International Computer Measurement Group Conference*, 2002, pp. 667–674.

[147] ——, "More new software antipatterns: Even more ways to shoot yourself in the foot," in *29th International Computer Measurement Group Conference*, 2003, pp. 717–725.

[148] ——, "Software performance engineering," in *UML for Real - Design of Embedded Real-Time Systems*. Kluwer, 2003, pp. 343–365.

[149] ——, "Software performance antipatterns for identifying and correcting performance problems," in *CMG'12*, 2012.

[150] K. Sneha and G. M. Malle, "Research on software testing techniques and software automation testing tools," in *International Conference on Energy, Communication, Data Analytics and Soft Computing*, 2017, pp. 77–81.

[151] J. Spel, S. Junges, and J.-P. Katoen, "Finding provably optimal Markov chains," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2021, pp. 173–190.

[152] I. Stefanakos, R. Calinescu, and S. Gerasimou, "Probabilistic program performance analysis," in *47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2021, pp. 148–157.

[153] W. J. Stewart, "Performance modelling and Markov chains," in *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. Springer Berlin Heidelberg, 2007, pp. 1–33.

[154] ——, *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, 2009.

[155] TIOBE. TIOBE Index for May 2022: Most used programming languages. [Online]. Available: https://www.tiobe.com/tiobe-index/

[156] M. Tribastone, S. Gilmore, and J. Hillston, "Scalable differential analysis of process algebra models," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 205–219, 2012.

[157] K. S. Trivedi and A. Bobbio, *Reliability and Availability Engineering - Modeling, Analysis, and Applications*. Cambridge University Press, 2017.

[158] C. Trubiani, "Automated generation of architectural feedback from software performance analysis results," Ph.D. dissertation, University of L'Aquila, 2011.

[159] C. Trubiani, A. Aleti, S. Goodwin, P. Jamshidi, A. van Hoorn, and S. Gratzl, "Vis-arch: Visualisation of performance-based architectural refactorings," in *Software Architecture*. Springer International Publishing, 2020, pp. 182–190.

[160] C. Trubiani, A. Ghabi, and A. Egyed, "Exploiting traceability uncertainty between software architectural models and extra-functional results," *Journal of Systems and Software*, vol. 125, pp. 15–34, 2017.

[161] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2011, p. 19–30.

[162] C. Trubiani, A. Koziolek, V. Cortellessa, and R. Reussner, "Guilt-based handling of software performance antipatterns in palladio architectural models," *Journal of Systems and Software*, vol. 95, pp. 141–165, 2014.

[163] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *ICPE*, 2012, pp. 247–248.

[164] W. Visser and C. S. Pasareanu, "Probabilistic programming for Java using symbolic execution and model counting," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017*, 2017, pp. 35:1–35:10.

[165] L. Wasserman, "Random variables," in *All of Statistics: A Concise Course in Statistical Inference*. Springer New York, 2004, pp. 19–46.

[166] W. Wei and B. Selman, "A new approach to model counting," in *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, 2005, pp. 324–339.

[167] A. Wert, J. Happe, and L. Happe, "Supporting swift reaction: automatically uncovering performance problems by systematic experiments," in *ICSE'13*, 2013, pp. 552–561.

[168] D. Weyns, "Software engineering of self-adaptive systems," in *Handbook of Software Engineering*. Springer International Publishing, 2019, pp. 399–443.

[169] G. A. Young and R. L. Smith, *Essentials of Statistical Inference*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2005.

[170] X. Yu, S. Han, D. Zhang, and T. Xie, “Comprehending performance from real-world execution traces: A device-driver case,” in *ACM SIGPLAN Notices*, vol. 49, 2014, pp. 193–206.

[171] J. Zhang, Y. Lu, K. Shi, and C. Xu, “Empirical research on the application of a structure-based software reliability model,” *IEEE/CAA Journal of Automatica Sinica*, pp. 1–10, 2020.