RESEARCH PROSPECT

Need a dissertation on a similar topic? Check out how our dissertation services can help you.

Dipartimento di Informatica
Università di L'Aquila
SEA Group
Via Vetoio, I-67100 L'Aquila, Italy
http://www.di.univaq.it

PhD Thesis in Computer Science

Automated generation of architectural feedback from software performance analysis results

Catia Trubiani

2011

PhD Advisor
Prof. Vittorio Cortellessa

© Catia Trubiani, 2011. All rights reserved

Happiness only real when shared

ABSTRACT

Over the last decade, research has highlighted the importance of integrating the performance analysis in the software development process. Software Performance Engineering (SPE) has been recognized as the discipline that represents the entire collection of engineering activities, used throughout the software development cycle, and directed to meet performance requirements. Performance is in fact an essential quality attribute of every software system, it is a complex and a pervasive property difficult to understand. If performance targets are not met, a variety of negative consequences (such as damaged customer relations, business failures, lost income, etc.) can impact on a significant fraction of projects. Performance problems cause delays, failures on deployment, redesigns, even a new implementation of the system or abandonment of projects, which lead to significant costs.

All these factors motivate the activities of modeling and analyzing the performance of software systems at the earlier phases of the lifecycle by reasoning on predictive quantitative results in order to avoid an expensive rework, possibly involving the overall software system. To this purpose, many model-based performance analysis techniques have been successfully proposed. Nevertheless, the problem of interpreting the results of performance analysis is still critical in the software performance domain: mean values, variances, and probability distributions are hard to interpret for providing feedback to software architects. Support to the interpretation of performance analysis results that helps to fill the gap between numbers and architectural alternatives is still lacking.

The aim of this thesis is to provide an automated feedback to make the performance analysis results usable at the software architectural level. We devise a methodology to keep track of the performance knowledge that usually tends to be fragmented and quickly lost. The purpose is to interpret the performance analysis results and to suggest the most suitable architectural reconfigurations, while the development progresses. The framework we propose is aimed at addressing this problem with performance antipatterns that are recurring solutions to common mistakes (i.e. bad practices) affecting performance. Such antipatterns can play a key role in the software performance domain, because they can be used in the search of performance problems as well as in the formulation of solutions in terms of architectural alternatives. The approach we propose is validated with two case studies: (i) E-Commerce system, modeled with UML, where the performance model has been analytically solved; (ii) Business Reporting system, modeled with the Palladio Component Model, where the performance analysis has been conducted through simulation. Experimental results finally demonstrate its applicability and validity.

Key words: Software Architecture, Performance Evaluation, Antipatterns, Feedback Generation, Architectural Alternatives, Unified Modeling Language (UML), Palladio Component Model (PCM).

ACKNOWLEDGMENTS

It is a great pleasure to thank the many people that supported me during my Ph.D. studies.

Words will never be enough to express my gratitude to my advisor Vittorio Cortellessa for his constant and precious guidance throughout my doctoral program, for the time spent on discussing and accurately reviewing all the research work of this thesis. Besides his scientific competence, I enjoyed a lot his positive attitude to make a friendly work atmosphere. I learned much more than scientific concepts from him, and his enthusiasm has been fundamental to overcome all the difficulties which arose in these three years. Thanks also for all the opportunities to meet important people in the research field, I am really grateful for the visit of Dorina Petriu in our department in L'Aquila.

I would like to thank the external reviewers of this thesis, Ralf Reussner and Catalina M. Lladó, for their careful reading and for their valuable comments and suggestions.

Unreserved thanks go to Antinisca Di Marco for her fundamental scientific contribution to this manuscript. She was very closed to me, I learnt really a lot from our discussions, and her encouragement was essential to believe in this research project.

Sincere thanks go to Raffaela Mirandola that showed credit in this work and my skills. She is a special person, I like her nice attitude to make me feel at ease every time we meet.

I really appreciated the opportunity to collaborate with the Software Design and Quality research group headed by Ralf Reussner at the Karlsruhe Institute of Technology. I found very valuable colleagues with whom I shared work and entertainment. In particular, it was a great pleasure to work with Anne, I enjoyed our long discussions that always pointed out more questions than answers. I especially thank Lucia for her close friendship and we had a lot of laughs together, Zoya to listen my troubles and to tell me her life experiences, Heiko and Jens to involve me in barbecue events, Fabian and Niko to help me with the strange german words of the coffee machine. Thanks to you all for the funny evenings.

I want to thank the Computer Science department, in particular the SEA Group, I think that all of you are special persons. I want to especially thank Alfonso Pierantonio and Romina Eramo with whom I had the chance to work with, and they gave me many useful and meaningful advices. Of course I cannot forget my roommates Alessandro, Luca, Romina E., Romina S., and Vasco whose pleasant company made easier all the obstacles along the way. They greatly alleviated the stress experienced while writing this thesis.

Many thanks to my past roommates in Rome. I would like to name Francesca L.P., Giovanni, Andrea, Alessandro, Lorenzo, Dario, Francesca M., Vincenzo, Simone, Saverio at "Tor Vergata". I would like to name Paolo, Gabriele, Ugo, Silvia, Ilaria at "La Sapienza". I am really grateful for their support especially after the earthquake in L'Aquila when

thirty seconds threw into confusion everyone and everything seemed insurmountable. Thanks for offering me a desk and a nice working environment at that time.

Many thanks to my flatmates Fabiana, Valentina, Alessandro, Davide, and Nico with whom I enjoyed the wonderful experience to live in L'Aquila. Thanks also to my past Rome flatmates Francesco, Mariangela, and Valentina who sustain me every time.

A special thank to my travelling friends with whom I cross the Gran Sasso mountain each day. In particular I want to thank Sergio, Federica, Chiara, Gabriele, and Francesca that make all travels less boring than expected. Our complaints towards public transportation (delays, strikes, etc.) often end in front of a cappuccino and a chocolate croissant.

Best thanks to my always friends: Marina, Sara, Simona, Valeria. Although I am often far from them, they are in everything I do. They are the first persons with whom I share all my defeats and victories, and their support is indispensable to keep me smiling. To my recent, but not less important, friends: Fabiana and Anna with whom I share the membership of the jellyfish trio, thanks to let me turn off the brain from the work during the weekends.

Many thanks to my secondary school friends that even remotely make me feel never alone. I would like to especially thank Egidio with whom I shared all my fears to be phD students in a country like Italy where meritocracy is only claimed but not applied, and Fabio with whom I share all my birthdays, he always remembers me that I am three days older than him and I think it is the right motivation to let him paying for both celebrations.

Last but not least, my deepest gratitude goes to my family. To Roberta and Sara for their continuous presence, to Tiziana and Valeria for their uncommon strength, to my little nephew Gianluca that makes me forget all my worries with his joyful smile. Finally, I am eternally grateful to my parents for their unflagging love and support throughout my life.

Many Thanks to all of you, this thesis would have no sense without any person I named!

TABLE OF CONTENTS

Abstract **v**

Acknowledgments **vii**

Table of Contents **viii**

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation 2
1.2 Problem statement 3
1.3 Thesis Focus and Key Contributions 6
1.4 Thesis Outline 9

2 Software Performance Feedback: state of the art **11**

2.1 Antipattern-based Approaches 11
2.2 Rule-based Approaches 13
2.3 Search-based Approaches 14
2.3.1 Design Space Exploration Approaches 15
2.3.2 Metaheuristic Approaches 15
2.4 Summary 16

3 Performance Antipatterns **19**

3.1 Introduction 19
3.2 Background: textual description 21
3.3 A graphical representation 23
3.3.1 Single-value Performance Antipatterns 23
3.3.2 Multiple-values Performance Antipatterns 42
3.4 A logic-based specification of the antipattern problem 49
3.4.1 Single-value Performance Antipatterns 50
3.4.2 Multiple-values Performance Antipatterns 63
3.4.3 Summary 66
3.5 Towards the specification of the antipattern solution 69

4 Specifying Antipatterns: a Model-Driven Approach **75**

4.1 Performance Antipatterns Modeling Language (PAML) 75
4.2 A model-based specification of the antipattern problem 80

4.2.1 Single-value Performance Antipatterns 80
4.2.2 Multiple-values Performance Antipatterns 97
4.2.3 Summary 100
4.3 Towards the specification of a model-based framework 103

5 Detecting and Solving Antipatterns in Concrete Modeling Languages 109
5.1 UML and Marte profile 109
5.1.1 Foundations 109
5.1.2 Detecting antipatterns with the mapping of PAML onto UML . . 111
5.1.3 Solving antipatterns in UML 116
5.2 Palladio Component Model 118
5.2.1 Foundations 119
5.2.2 Detecting antipatterns with the mapping of PAML onto PCM . . . 121
5.2.3 Solving antipatterns in PCM 124
5.3 Summary and Lessons Learned 127

6 Validation 131
6.1 A case study in UML 131
6.1.1 E-commerce System 132
6.1.2 Detecting Antipatterns 136
6.1.3 Solving Antipatterns 139
6.1.4 Experimentation 141
6.2 A case study in PCM 142
6.2.1 Business Reporting System 144
6.2.2 Detecting Antipatterns 145
6.2.3 Solving Antipatterns 146
6.2.4 Experimentation 147

7 A step ahead in the antipatterns solution 151
7.1 A strategy to identify ”guilty” Performance Antipatterns 151
7.2 An approach for ranking antipatterns 152
7.2.1 Violated Requirements 153
7.2.2 Complete Antipatterns List 154
7.2.3 Filtering antipatterns 155
7.2.4 Ranking antipatterns 156
7.3 Experimenting the approach 158
7.3.1 Case study 158
7.3.2 Experimental results 159
7.4 Discussion 164
7.5 Towards the simultaneous solution of antipatterns 165

8 Further steps in the antipatterns solution 169
8.1 Workload sensitivity analysis 169
8.2 Stakeholders analysis 170
8.3 Operational profile analysis 172
8.4 Cost analysis 173

9 Conclusion **175**
9.1 Achievements 176
9.2 Open issues and Future works 177
9.2.1 Short term goals 177
9.2.2 Long term goals 178

References **180**

A An XML Schema for performance antipattern elements **191**
A.1 Static View 193
A.2 Dynamic View 197
A.3 Deployment View 200
A.4 Conclusion 203

B Modeling Notations **205**
B.1 Software Architectural Model 205
B.1.1 Automata 205
B.1.2 Process Algebras 206
B.1.3 Petri Nets 207
B.1.4 Message Sequence Charts 208
B.1.5 Use Case Maps 208
B.2 Performance Model 209
B.2.1 Markov Processes 209
B.2.2 Queueing Networks 210
B.2.3 Stochastic Process Algebras 211
B.2.4 Stochastic Timed Petri Nets 212
B.2.5 Simulation Models 213

LIST OF FIGURES

1.1 Automated software performance process. 4
1.2 Automated software performance process by introducing the antipatterns. 6
1.3 The main activities of the PANDA framework. 7
1.4 An antipattern-based process for the software performance process. . . . 8

3.1 A first approach of the Results Interpretation & Feedback Generation steps. 20
3.2 A graphical representation of the *Blob-controller* Antipattern. 25
3.3 A graphical representation of the *Blob-dataContainer* Antipattern. 26
3.4 A graphical representation of the *Concurrent Processing Systems* Antipattern. 28
3.5 A graphical representation of the *Pipe and Filter Architectures* Antipattern. 30
3.6 A graphical representation of the *Extensive Processing* Antipattern. 32
3.7 A graphical representation of the *Circuitous Treasure Hunt* Antipattern. . 34
3.8 A graphical representation of the *Empty Semi Trucks* Antipattern. 37
3.9 A graphical representation of the *Tower of Babel* Antipattern. 39
3.10 A graphical representation of the *One-Lane Bridge* Antipattern. 41
3.11 A graphical representation of the *Excessive Dynamic Allocation* Antipattern. 43
3.12 A graphical representation of the *Traffic Jam* Antipattern. 45
3.13 A graphical representation of *The Ramp* Antipattern. 47
3.14 A graphical representation of the *More Is Less* Antipattern. 48
3.15 Bird's-eye look of the XML Schema and its views. 50
3.16 A graphical representation of the reconfiguration actions for solving the *Blob-controller* Antipattern. 72

4.1 The Performance Antipattern Modeling Language (PAML) structure. . . . 76
4.2 The Enriched Software Modeling Language (SML+). 77
4.3 PAML implementation in the Eclipse platform. 78
4.4 PAML Model wizard for creating new instances. 78
4.5 PAML Editor properties. 79
4.6 PAML-based model of the *Blob* Antipattern. 81
4.7 PAML-based model of the *Concurrent Processing Systems* Antipattern. . 83
4.8 PAML-based model of the *Pipe and Filter Architectures* Antipattern. . . . 85
4.9 PAML-based model of the *Extensive Processing* Antipattern. 87
4.10 PAML-based model of the *Circuitous Treasure Hunt* Antipattern. 89
4.11 PAML-based model of the *Empty Semi Trucks* Antipattern. 91
4.12 PAML-based model of the *Tower of Babel* Antipattern. 93

4.13 PAML-based model of the *One-Lane Bridge* Antipattern. 94
4.14 PAML-based model of the *Excessive Dynamic Allocation* Antipattern. . . 96
4.15 PAML-based model of the *Traffic Jam* Antipattern. 98
4.16 PAML-based model of the *The Ramp* Antipattern. 99
4.17 PAML-based model of the *More Is Less* Antipattern. 101
4.18 Translating antipatterns into concrete modeling languages. 104
4.19 Metamodel instantiation via weaving models. 104
4.20 Weaving model over different software modeling languages. 105
4.21 Tranforming PAML-based models in OCL queries. 106
4.22 Feedback Generation approach by means of model-driven techniques. . . 106

5.1 UML Compliance Level 3 top-level package merges. 110
5.2 Informal description of the MARTE dependencies with other OMG standards. 111
5.3 Artifacts of a PCM instance. 119
5.4 Example of a PCM model. 120
5.5 An excerpt of the MARTE *RunTimeContext* package. 129

6.1 ECS case study: customized software performance process. 131
6.2 ECS case study: Use Case Diagram. 132
6.3 ECS (annotated) software architectural model. 134
6.4 ECS - Queueing Network model. 135
6.5 ECS- the *Blob* antipattern occurrence. 137
6.6 ECS- the *Concurrent Processing Systems* antipattern occurrence. 137
6.7 ECS- the *Empty Semi Trucks* antipattern occurrence. 138
6.8 ECS model refinement: reiteration of the software performance process. . 140
6.9 BRS case study: customized software performance process. 142
6.10 Screenshot of the PCM Bench extension providing the usage of antipatterns knowledge. 143
6.11 PCM software architectural model for the BRS system. 144
6.12 PCM usage model for the BRS system. 145
6.13 Response time of the $system$ across the iterations of the antipattern-based process. 148
6.14 Summary of the process for the BRS system. 148

7.1 A process to improve the performance analysis interpretation. 152
7.2 BRS software architectural model. 158
7.3 RT($CS_{graphicalReport}$) vs the guiltness degree of antipatterns. 163
7.4 RT($CS_{onlineReport}$) vs the guiltness degree of antipatterns. 163
7.5 T($CS_{graphicalReport}$) vs the guiltness degree of antipatterns. 164
7.6 How to decide between different moves. 165
7.7 Process for the combination of performance antipatterns. 166

8.1 Workload sensitivity analysis as a support for solving antipatterns. 170
8.2 Stakeholders analysis as a support for solving antipatterns. 171
8.3 Operational profile analysis as a support for solving antipatterns. 172

8.4 Cost analysis as a support for solving antipatterns. 173

A.1 An excerpt of the XML Schema. 192
A.2 XML Schema - *Static View*. 194
A.3 An example of the *Relationship* element. 196
A.4 An example of the *StructuredResourceDemand* element. 196
A.5 An example of the *PerformanceMetrics* element. 197
A.6 XML Schema - *Dynamic View*. 198
A.7 An example of the *Behavior* element. 199
A.8 XML Schema - *Deployment View*. 201
A.9 An example of the *NetworkLink* element. 202
A.10 An example of the *ProcesNode* element. 203

B.1 A simple example of the *Automata* modeling notation. 206
B.2 A simple example of the *Petri Nets* modeling notation. 207
B.3 Basic Symbols of the *Use Case Maps* modeling notation. 209
B.4 A simple example of the *Æmilia* modeling notation. 211

List of Tables

2.1 Summary of the approaches related to the generation of architectural feedback. . . . 17

3.1 Performance Antipatterns: problem and solution [123]. . . . 22
3.2 Functions specification. . . . 67
3.3 Thresholds specification: software characteristics. . . . 68
3.4 Thresholds specification: hardware characteristics. . . . 68
3.5 Thresholds specification: requirements. . . . 69
3.6 Thresholds specification: slopes. . . . 69
3.7 A logic-based representation of Performance Antipatterns. . . . 70

4.1 A metamodel-based representation of Performance Antipatterns. . . . 102

5.1 SML+ Static View Model Elements and their mapping to UML metamodel elements. . . . 112
5.2 SML+ Dynamic View Model Elements and their mapping to UML metamodel elements. . . . 113
5.3 SML+ Deployment View Model Elements and their mapping to UML metamodel elements. . . . 114
5.4 SML+ Static View Model Elements and their mapping to PCM metamodel elements. . . . 121
5.5 SML+ Dynamic View Model Elements and their mapping to PCM metamodel elements. . . . 122
5.6 SML+ Deployment View Model Elements and their mapping to PCM metamodel elements. . . . 123
5.7 Performance Antipatterns detectable and solvable in UML and PCM. . . . 128

6.1 Input parameters for the queueing network model in the ECS system. . . . 133
6.2 Response time requirements for the ECS software architectural model. . . . 135
6.3 ECS- antipatterns boundaries binding. . . . 136
6.4 ECS Performance Antipatterns: problem and solution. . . . 139
6.5 Input parameters for the queueing network model across different software architectural models. . . . 140
6.6 Response time required and observed. . . . 141
6.7 BRS Performance Antipatterns: problem and solution. . . . 146
6.8 Response time of the $system$ across BRS software architectural model candidates. . . . 149

7.1 Example of Performance Requirements. 153
7.2 Details of Violated Requirements. 154
7.3 Complete Antipatterns List. 155
7.4 Filtered Antipatterns List. 155
7.5 How to rank performance antipatterns 156
7.6 BRS - Performance requirement analysis 159
7.7 BRS - Violated Requirements 160
7.8 BRS- Complete Antipatterns List 160
7.9 BRS - Ranked Antipatterns List 161
7.10 RT($CS_{graphicalReport}$) across different software architectural models 162

CHAPTER 1

INTRODUCTION

Over the last decade, research has highlighted the importance of integrating non-functional analysis activities in the software development process, in order to meet non-functional requirements. Among these, performance is one of the most influential factors to be considered since performance problems may be so severe that they can require considerable changes at any stage of the software lifecycle, in particular at the software architecture level or design phase and, in the worst cases, they can even impact the requirements level.

In the software development process it is fundamental to understand if performance requirements are fulfilled, since they represent what end users expect from the software system, and their unfulfillment might produce critical consequences. The early development phases may heavily affect the quality of the final software product, and wrong decisions at early phases may imply an expensive rework, possibly involving the overall software system. Therefore, performance issues must be discovered early in the software development process, thus to avoid the failure of entire projects [74, 75].

In the research community there is a growing interest in the early validation of performance requirements. The model-based approach, pioneered under the name of Software Performance Engineering (SPE) by Smith [117, 131, 116] creates performance models early in the development cycle and uses quantitative results from these models to adjust the architecture and design [40] with the purpose of meeting performance requirements [133]. Software architectures [63, 41] have emerged as a foundational concept for the successful development of large, complex systems, since they support five aspects of the software development: understanding, reuse, evolution, analysis and management [62].

The Software Architecture is the earliest model of a software system created along the lifecycle. Perry and Wolf in [109] defined the architecture as the selection of architectural elements, their interactions, and the constraints on those elements. Garlan and Shaw in [64] defined the architecture as a collection of computational components together with a description of the interactions between these components. In this thesis we adopt the latter definition.

Several approaches have been successfully applied by modeling and analyzing the performance of software systems on the basis of predictive quantitative results [19, 89, 135,

134, 111]. However we are still far from considering the performance analysis as an integrated activity into the software development that effectively supports all the phases of the software lifecycle. In practice it is generally acknowledged that the lack of performance requirement validation during the software development process is mostly due to the knowledge gap between software engineers/architects and performance experts (as special skills are required) rather than to foundational issues. Moreover, short time to market constraints make this situation even more critical.

The problem of interpreting the performance analysis results is still quite critical, since it is difficult to understand mean values, variances, and probability distributions. Additionally, a large gap exists between the representation of performance analysis results and the feedback expected by software architects. The former usually contains numbers (such as mean response time, throughput variance), whereas the latter should embed architectural suggestions useful to overcome performance problems (such as split a software component in two components and re-deploy one of them). Such activities are today exclusively based on the analysts' experience, and therefore their effectiveness often suffers the lack of automation. In this scenario, we believe that the automated generation of feedback may work towards the problem solution, since it has the important role of making performance analysis results usable at the software architectural level. It means, for example, that from a bad throughput value, it is possible to identify the software components and/or interactions responsible for that bad value.

1.1 MOTIVATION

Software performance is a pervasive quality difficult to understand, because it is affected by every aspect of the design, code, and execution environment [133]. By conventional wisdom performance is a serious problem in a significant fraction of projects. Performance failures occur when a software product is unable to meet its overall objectives due to inadequate performance. Such failures negatively impact the projects by increasing costs, decreasing revenue or both. Furthermore they cause delays, cost overruns, failures on deployment, and even abandonment of projects, as documented in the following, quoted from [122]:

"*Consider, as an example, that 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.*"

Performance management is a hot topic for the Information Technology (IT) industry and media, and it is interesting to know to what extent real organizations are getting to grips with the performance issues. A recent survey [1] has been conducted on the performance management practice inviting 150 senior IT managers (i.e. responsible for testing and performance working at large organizations across Europe) to fill a questionnaire about their experiences in the field. The survey demonstrates that performance failures are still occurring in many organizations, as more than half of organizations said they experience unexpected performance issues in 20% or more of their newly deployed applications.

The primary cause of performance failures is a reactive approach to performance during the development process. Cost and schedule pressures encourage project managers to adopt a *fix-it-later* approach in which performance is ignored until there is a problem. When a problem is discovered, more hardware is needed, developers must try to tune the software to meet performance objectives or both, but in some cases it is simply not possible to meet performance objectives by tuning.

The fundamental lesson learned from the documented projects is that it is better to prevent performance failures and the resulting project crises. A proactive approach to software performance management has the benefit of anticipating probable performance problems, because it is based on techniques for identifying and responding to those problems early in the process. The goal is to produce a software that meets performance objectives, thus to avoid project crises due to the late discovery of performance issues.

1.2 Problem statement

Figure 1.1 schematically represents the typical steps that are executed at a generic phase of the software lifecycle to conduct a model-based performance analysis process. Rounded boxes in the Figure represent operational steps whereas square boxes represent input/output data. Vertical lines divide the process in three different phases: in the *modeling* phase an (annotated) software architectural model is built; in the *performance analysis* phase a performance model is obtained through model transformation, and such model is solved to obtain the performance results of interest; in the *refactoring* phase the performance results are interpreted and, if necessary, feedback is generated as refactoring actions on the original software architectural model.

A *software architectural model* (see Figure 1.1) is an abstract view of the software system. Complementary types of model provide different system information. Such different models present the system from different perspectives, such as external perspective showing the system's context or environment, behavioral perspective showing the behavior of the system, etc. [125]. We refer to *(annotated)* models, since annotations are meant to add information that led to execute performance analysis such as the incoming workload to the system, service demands, hardware characteristics, etc. There exists many notations to describe all these aspects of a software system (e.g. automata, process algebras, petri nets and process algebras), surveyed in [19] and shortly reported in Appendix B.1.

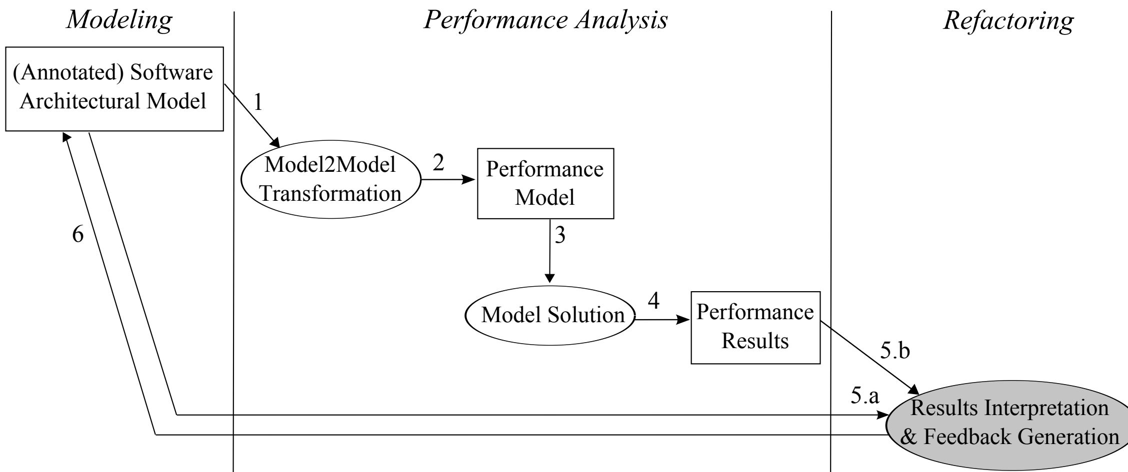


Figure 1.1: Automated software performance process.

A *performance model* (see Figure 1.1) describes how system operations use resources, and how resource contention affects operations. The solution of a performance model supports the prediction of the properties of a system before it is built, or the effect of a change before it is carried out. This gives a special warning role to early modeling. However as the implementation proceeds, more accurate models can be created by other means, and may have additional uses, in particular: (i) design of performance tests; (ii) configuration of products for delivery; (iii) evaluation of planned evolutions of the design, while recognizing that no system is ever final [133]. There exists many notations to describe all these aspects of a software system (e.g. queueing networks, layered queues, stochastic petri nets and process algebras), surveyed in [19] and shortly reported in Appendix B.2.

Performance results (see Figure 1.1) refer to the following performance indices [81]:

- *Response time* is defined as the time interval between a user request of a service and the response of the system. Usually, upper bounds are defined in "business" requirements by the end users of the system.
- *Utilization* is defined as the ratio of busy time of a resource and the total elapsed time of the measurement period. Usually, upper bounds are defined in "system" requirements by system engineers on the basis of their experience, scalability issues, or constraints introduced by other concurrent software systems sharing the same hardware platform.
- *Throughput* is defined as the rate at which requests can be handled by a system, and is measured in requests per time. Throughput requirements can be both "business" and "system" requirements, depending on the target it applies; for the same motivation it can represent either an upper or a lower bound.

The modeling and performance analysis phases (i.e. arrows numbered from 1 through 4) represent the forward path from an (annotated) software architectural model all the

way through the production of performance results of interest. While in this path quite well-founded approaches have been introduced for inducing automation in all steps (e.g. [19, 89, 137]), there is a clear lack of automation in the backward path that shall bring the analysis results back to the software architecture.

The core step of the backward path (i.e. the shaded box of Figure 1.1) is where the performance analysis results have to be interpreted in order to detect, if any, performance problems; once performance problems have been detected (with a certain accuracy) somewhere in the model, solutions have to be applied to remove those problems[1]. A performance problem originates from a set of unfulfilled requirement(s), such as the estimated response time of a service is higher than the required one. If all the requirements are satisfied then the feedback obviously suggests no changes.

In Figure 1.1, the (annotated) software architectural model (label 5.a) and the performance results (label 5.b) are inputs to the core step that searches problems in the model. The search of performance problems in architectural models may be quite complex and needs to be smartly driven towards the problematic areas of the model. The complexity of this step stems from several factors:

(i) performance indices are basically numbers and often they have to be jointly examined: a single performance index (e.g. the utilization of a service) might not be enough to localize the critical parts of a software architecture, since a performance problem might emerge only if other indices (e.g. the throughput of a neighbor service) are analyzed;

(ii) performance indices can be estimated at different levels of granularity (e.g. the response time index can be evaluated at the level of a cpu device, or at the level of a service that spans on different devices) and it is unrealistic to keep under control all indices at all levels of abstraction;

(iii) software architectural models can be quite complex, and the origin of performance problems emerges only looking at the architectural elements described in different views of a system (such as static structure, dynamic behavior, deployment configurations, etc.).

The problem tackled in this thesis is to interpret performance analysis results and to introduce automation in the backward path (label 6 of Figure 1.1) in the form of architectural alternatives (e.g. split a software component in two components and re-deploy one of them) that do not suffer of the original performance problems. Such automation provides several gains since it notifies the software architect of the interpretation of the analysis results. Performance issues and subsequent architectural emendations aimed at removing such issues are outlined without the intervention of performance experts. We believe that both the software architectural and performance models represent suitable instruments

[1]Note that this task very closely corresponds to the work of a physician: observing a sick patient (the model), studying the symptoms (some bad values of performance indices), making a diagnosis (performance problem), prescribing a treatment (performance solution).

that can be refined accordingly to the feedback we provide, while the development progresses, thus to act early in the software lifecycle.

1.3 THESIS FOCUS AND KEY CONTRIBUTIONS

The research activity of this thesis is focused on the core step of Figure 1.1, and in Figure 1.2 the most promising elements that can drive this search have been explicitly represented, i.e. *performance antipatterns* (input labeled 5.c to the core step). The rationale of using performance antipatterns is two-fold: on the one hand, a performance antipattern identifies a bad practice in the software architectural model that negatively affects the performance indices, thus to support the *results interpretation* step; on the other hand, a performance antipattern definition includes a solution description that lets the software architect devise refactoring actions, thus it supports the *feedback generation* step.

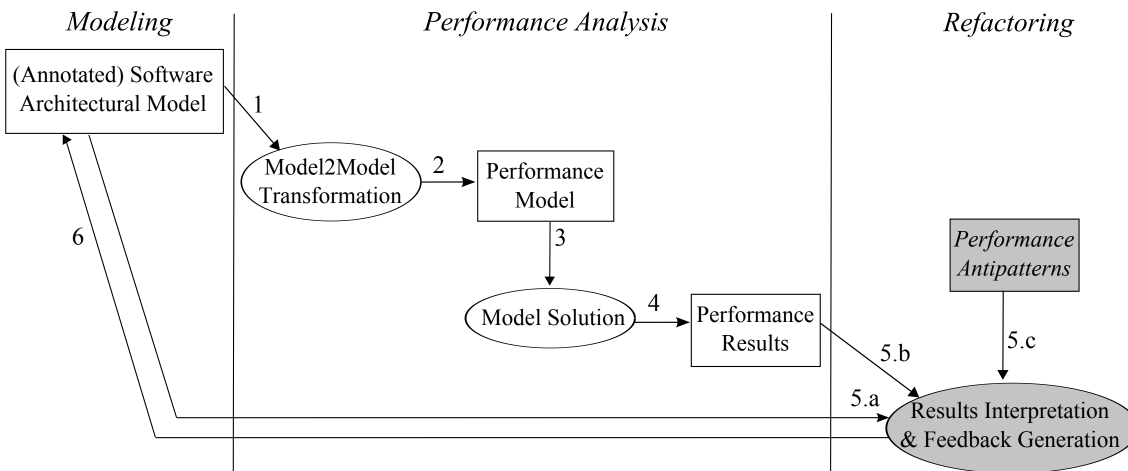


Figure 1.2: Automated software performance process by introducing the antipatterns.

The main reference we consider for performance antipatterns is the work done across the years by Smith and Williams [123] that have ultimately defined fourteen notation-independent antipatterns. Some other works present antipatterns that occur throughout different technologies, but they are not as general as the ones defined in [123] (more references are provided in Chapter 2).

The key contribution of this thesis is to address the problem of interpreting the performance results and generating architectural alternatives. This is achieved with a framework named PANDA: *P*erformance *A*ntipatterns a*N*d Fee*D*back in Software *A*rchitectures. The main activities performed within such framework are schematically shown in Figure 1.3: *specifying antipatterns*, to define in a well-formed way the properties that lead the software system to reveal a bad practice as well as the changes that provide a solution; *detecting antipatterns*, to locate antipatterns in software architectural models; *solving antipatterns*, to remove the detected performance problems with a set of refactoring actions that can be applied on the software architectural model.

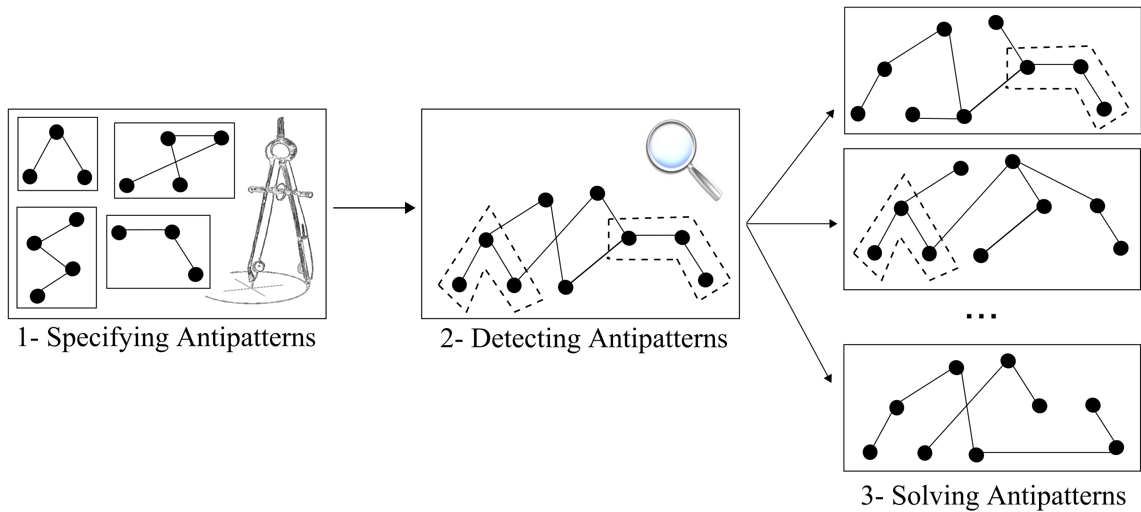


Figure 1.3: The main activities of the PANDA framework.

The activity of specifying antipatterns is performed by introducing a neutral and a coherent set of interrelated concepts to represent the software architectural model elements as well as the performance indices that occur in the definition of antipatterns (e.g. software resource, network resource utilization, etc.). Such activity is meant to be the basis for a machine-processable definition of antipatterns. An antipattern definition in fact includes: (i) the specification of the problem, i.e. a set of *antipattern-based rules* that interrogate the system elements to look for occurrences of the corresponding antipattern; (ii) the specification of the solution, i.e. a set of *antipattern-based actions* that can be applied on the system elements to remove a performance problem.

The activities of detecting and solving antipatterns are performed by respectively defining the antipattern-based rules and actions into concrete modeling notations. In fact, the modeling language used for the system is of crucial relevance, since the antipatterns (neutral) concepts must be translated into actual concrete modeling languages. The framework currently considers two notations: a system modeling language such as UML [12], and the Palladio Component Model (PCM) [22], i.e. a domain specific modeling language.

Figure 1.4 details the software performance modeling and analysis process of Figure 1.2. The core step is explicitly represented in two main operational steps: (i) *detecting antipatterns* that provides the localization of the critical parts of software architectural models thus to address the results interpretation problem; (ii) *solving antipatterns* that suggests the changes to be applied to the architectural model under analysis, thus to address the feedback generation problem.

Several *iteration*s can be conducted to find the software architectural model that best fits the performance requirements, since several antipatterns may be detected in an architectural model, and several refactoring actions may be available for solving each antipattern. At each iteration the actions we propose are aimed at building a new software architectural model, i.e. a *Candidate* (see Figure 1.4), that replaces the one under analysis. For

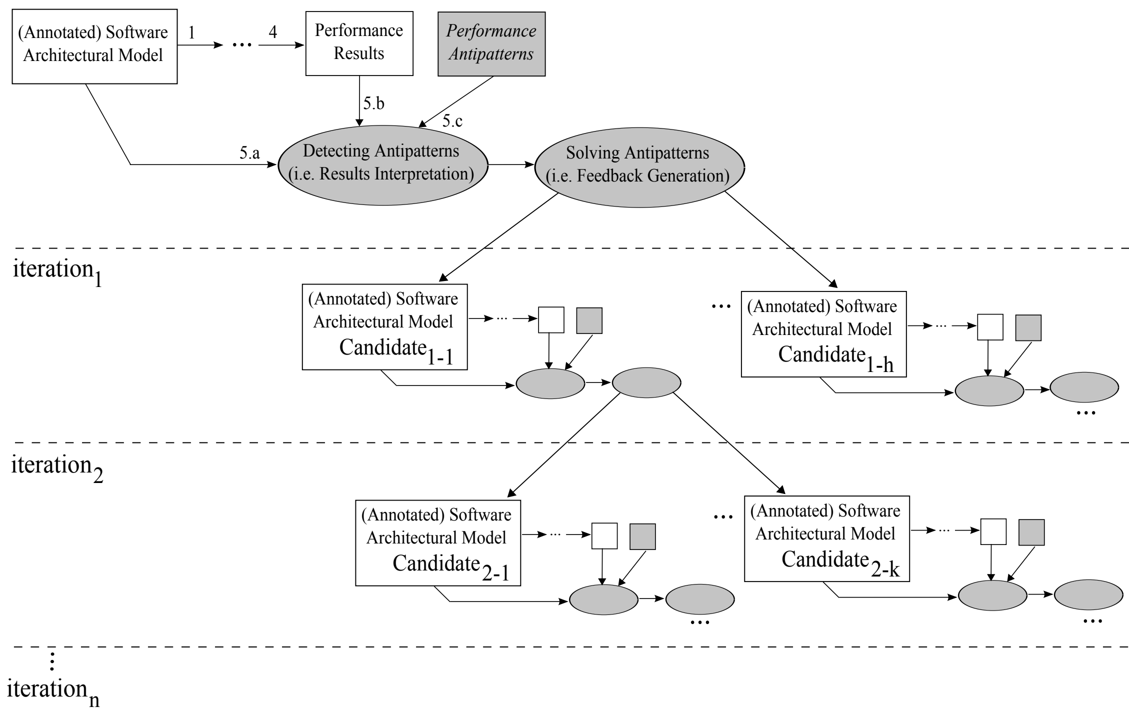


Figure 1.4: An antipattern-based process for the software performance process.

example, the (annotated) software architectural model $Candidate_{i-j}$ is generated at the i-th iteration and it denotes the j-th candidate for the iteration it belongs to. Then, the detection and solution approach can be iteratively applied to all newly generated candidates to further improve the system, if necessary.

Different termination criteria can be defined in the antipattern-based process:

(i) *fulfilment* criterion, i.e. all requirements are satisfied and a suitable software architectural model is found;

(ii) *no-actions* criterion, i.e. no antipatterns are detected in the software architectural models therefore no refactoring actions can be experimented;

(iii) *#iterations* criterion, i.e. the process can be terminated if a certain number of iterations have been completed.

It is worth to notice that the solution of one or more antipatterns does not a priori guarantee performance improvements, because the entire process is based on heuristic evaluations. However, an antipattern-based refactoring action is usually a correctness-preserving transformation that improves the quality of the software. For example, the interaction between two components might be refactored to improve performance by sending fewer messages with more data per message. This transformation does not alter the semantics of the application, but it may improve the overall performance.

We can conclude that performance antipatterns represent a promising instrument to introduce automation in the backward path of the software performance cycle. The benefit of using antipatterns is that they are descriptions of problems commonly encountered by performance engineers in practice. Such knowledge is acquired while analyzing complex software systems and exploited at the software architectural level.

1.4 THESIS OUTLINE

This thesis is organized as follows.

Chapter 2 reviews the main existing approaches in the research area referring to the generation of the architectural feedback. In particular, three categories of approaches are outlined: (i) antipattern-based approaches that make use of antipatterns knowledge to cope with performance issues; (ii) rule-based approaches that define a set of rules to overcome performance problems; (iii) search-based approaches that explore the problem space by examining options to deal with performance flaws. In the context of the search-based process two techniques can be applied: design space exploration that blindly examine all architectural alternatives, and metaheuristic search techniques (e.g. genetic algorithms) that search for local changes in the architectural model.

Chapter 3 describes the performance antipatterns we consider for the generation of the architectural feedback. A classification of antipatterns (i.e. single-value, multiple-values antipatterns) is discussed on the basis of the performance analysis (i.e. analytic method, simulation) results that we need to capture the bad practices. A graphical representation supports the interpretation of the natural language-based definitions of antipatterns, followed by a logic-based specification that better formalizes such interpretation. A structured description of the architectural model elements that occur in the definitions of antipatterns is provided in Appendix A, and it represents a groundwork for the definition of antipatterns as logical predicates.

Chapter 4 gives an overview of how model-driven engineering techniques can be applied to tackle the problem of automating the interpretation of performance analysis results and the generation of architectural feedback. A Performance Antipattern Modeling Language (PAML) is defined to build a user-friendly representation of antipatterns. They become here models conform to PAML, or in other words they are expressed by the concepts encoded in our modeling language. The antipatterns representation as PAML-based models allows to manipulate their (neutral) definition with model-driven techniques (e.g. weaving models) by deriving important features across different modeling notations.

Chapter 5 examines the performance antipatterns within the concrete modeling languages we consider for the validation of the approach. In particular, we discuss the semantic relations between PAML and UML and PCM metamodel elements respectively, in order to investigate the expressiveness of such notations. These experiences have allowed to classify the antipatterns in three categories: (i) detectable and solvable; (ii) semi-solvable

(i.e. the antipattern solution is only achieved with refactoring actions to be manually performed); (iii) neither detectable nor solvable.

Chapter 6 provides the experimentation on the UML and PCM modeling languages to validate the whole approach. Two case studies are analyzed: (i) the first one, i.e. an E-commerce system, is modeled with UML, transformed in a Queueing Network and solved with the analytic method; (ii) the second one, i.e. a Business Reporting system, is modeled with PCM, transformed in a Queueing Network and solved with simulation. In both cases our antipattern-based approach is applied: the detection of antipatterns leads to identify the most critical parts of software architectures; the solution of antipatterns gives rise to a set of architectural alternatives that actually demonstrate performance improvements for the case study under analysis.

Chapter 7 exploits the requirements analysis to achieve a step ahead in the antipatterns solution. Requirements are used to decide the most promising refactoring actions that can rapidly lead to remove performance problems. The antipatterns detected in software architectural models are ranked on the basis of their guiltiness for violated requirements in order to give a priority in the sequential solution of antipatterns, thus to quickly convey the desired result of requirements satisfaction. In this direction several interesting issues emerge, e.g. the simultaneous solution of multiple antipatterns is briefly discussed.

Chapter 8 collects the cross-cutting concerns (e.g. workload, operational profile, etc.) that influence the software performance analysis results and the generation of the architectural feedback in order to give a wide explanation of domain features that might emerge in this context. These features can be used to prioritize the solution of the detected antipatterns, thus to quickly achieve the requirements fulfillment.

Chapter 9 concludes the thesis. It gives a summary of the achieved results by pointing out the assumptions and the limitations from using the antipatterns in the software performance process. A list of open issues that remain uncovered in this thesis is provided and they represent the future works we intend to pursue.

CHAPTER 2

SOFTWARE PERFORMANCE FEEDBACK: STATE OF THE ART

In literature few related works can be found dealing with the interpretation of performance analysis results and the generation of architectural feedback. Most of them are based on monitoring techniques and therefore are conceived to only act after software deployment for tuning its performance. We are instead interested in model-based approaches that can be applied early in the software lifecycle to support design decisions.

In the following the main existing approaches for the automated generation of architectural feedback are surveyed. In particular, we identified three principal categories of approaches: (i) antipattern-based approaches (see Section 2.1) that make use of antipatterns knowledge to cope with performance issues; (ii) rule-based approaches (see Section 2.2) that define a set of rules to overcome performance problems; (iii) search-based approaches (see Section 2.3) that explore the problem space by examining options to deal with performance flaws. In the context of the search-based process two techniques can be applied: design space exploration (see Section 2.3.1) and metaheuristic (see Section 2.3.2).

2.1 ANTIPATTERN-BASED APPROACHES

The term *Antipattern* appeared for the first time in [33] in contrast to the trend of focus on positive and constructive solutions. Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences. Antipatterns have been applied in different domains. For example, in [128] data-flow antipatterns help to discover errors in workflows and are formalized through the CTL* temporal logic. As another example, in [31] antipatterns help to discover multi threading problems of Java applications and are specified through the LTL temporal logic.

Performance Antipatterns, as the name suggests, deal with performance issues of the software systems. They have been previously documented and discussed in different works: *technology-independent* performance antipatterns have been defined in [123]; *technology-specific* antipatterns have been defined in [52] and [127].

Williams et al. in [130] introduced the PASA (Performance Assessment of Software Architectures) approach. It aims at achieving good performance results [118] through a deep understanding of the architectural features. This is the approach that firstly introduces the concept of antipatterns as support to the identification of performance problems in software architectural models as well as in the formulation of architectural alternatives. However, this approach is based on the interactions between software architects and performance experts, therefore its level of automation is still low.

Cortellessa et al. in [43] introduced a first proposal of automated generation of feedback from the software performance analysis, where performance antipatterns play a key role in the detection of performance flaws. However, this approach considers a restricted set of antipatterns, and it uses informal interpretation matrices as support. Performance scenarios are described (e.g. the throughput is lower than the user requirement, and the response time is greater than the user requirement) and, if needed, some actions to improve such scenarios are outlined. The main limitation of this approach is that the interpretation of performance results is only demanded to the analysis of Layered Queue Networks (LQN) [110, 68, 69], i.e. a performance model. Such knowledge is not enriched with the features coming from the software architectural models, thus to hide feasible refactoring actions.

Enterprise technologies and EJB performance antipatterns are analyzed by Parsons et al. in [108]: antipatterns are represented as sets of rules loaded into a JESS [2] engine, and written in a Lisp-like syntax [100]. A rule-based performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However, it deals with Component-Based Enterprise Systems, targeting only Enterprise Java Bean (EJB) applications. It is based on the monitoring of the data from running systems, it extracts the run-time system design and detects EJB antipatterns by applying rules to it. Hence, the scope of [108] is restricted to such domain, and performance problems can neither be detected in other technology contexts nor in the early development stages.

By taking a wider look out of the performance domain, the management of *antipatterns* is a quite recent research topic, whereas there has already been a significant effort in the area of software design *patterns*. It is out of scope to address such wide area, but it is worth to mention some approaches dealing with patterns.

Elaasar et al. in [55] introduced a metamodeling approach to pattern specification. In the context of the OMGs 4-layer metamodeling architecture, the authors propose a pattern specification language (i.e. Epattern, at the M3 level) used to specify patterns in any MOF-compliant modeling language at the M2 layer.

France et al. in [59] introduced a UML-based pattern specification technique. Design patterns are defined as models in terms of UML metamodel concepts: a pattern model describes the participants of a pattern and the relations between them in a graphical notation by means of roles, i.e. the properties that a UML model element must have to match the corresponding pattern occurrence.

2.2 RULE-BASED APPROACHES

Barber et al. in [21] introduced heuristic algorithms that in presence of detected system bottlenecks provide alternative solutions to remove them. The heuristics are based on architectural metrics that help to compare different solutions. In a Domain Reference Architecture (DRA) the modification of functions and data allocation can affect non-functional properties (for example, performance-related properties such as component utilization). The tool *RARE* guides the derivation process by suggesting allocations based on heuristics driven by static architectural properties. The tool *ARCADE* extends the RARE scope by providing dynamic property measures. ARCADE evaluation results subsequently fed back to RARE can guide additional heuristics that further refine the architecture. However, it basically identifies and solve only software bottlenecks, more complex problems are not recognized.

Dobrzanski et al. in [51] tackled the problem of refactoring UML models. In particular, *bad smells* are defined as structures that suggest possible problems in the system in terms of functional and non-functional aspects. Refactoring operations are suggested in the presence of bad smells. Rules for refactoring are formally defined, and they take into account the following features: (i) cross integration of structure and behavior; (ii) support for component-based development via composite structures; and (iii) integration of action semantics with behavioral constructs. However, no specific performance issue is analyzed, and refactoring is not driven by unfulfilled requirements.

McGregor et al. in [101] proposed a framework (ArchE) to support the software designers in creating architectures that meet quality requirements. It embodies knowledge of quality attributes and the relation between the achievement of quality requirements and architectural design. It helps to create architectural models by collecting requirements (in form of scenarios) and the information needed to analyze the quality criteria for the requirements. It additionally provides the evaluation tools for modifiability or performance analysis. However, the suggestions (or tactics) are not well explained, and it is not clear at which extent the approach can be applied.

Kavimandan et al. in [85] presented an approach to optimize deployment and configuration decisions in the context of distributed, realtime, and embedded (DRE) component-based systems. Bin packing algorithms have been enhanced, and schedulability analysis have been used to make fine-grained assignments that indicate how components are allocated to different middleware containers, since they are known to impact on the system performance and resource consumption. However, the scope of this approach is limited to deployment and configuration features.

Xu in [138] presented an approach to software performance diagnosis that identifies performance flaws before the software system implementation. It defines a set of *rules* (specified with the Jess rule engine [2]) aimed at detecting patterns of interaction between resources. The method is applied to UML [12] that employ standard profiles, i.e. the SPT or Schedulability, Performance and Time profile [14] and its successor MARTE [13].

The software architectural models are translated in a performance model, i.e. Layered Queueing Networks (LQNs) [110, 68, 69], and then analyzed. The approach limits the detection to bottlenecks and long execution paths identified and removed at the level of the LQN performance model. The actions to solve the performance issues are: *change the configuration*, i.e. increase the size of a buffer pool or the amount of existing processors; and *change the design*, i.e. increase parallelism and splitting the execution of task in synchronous and asynchronous parts. The overall approach applies only to LQN models, hence its portability to other notations is yet to be proven and it may be quite complex.

2.3 Search-based Approaches

A wide range of different optimization and search techniques have been introduced in the field of Search-Based Software Engineering (SBSE) [70, 73], i.e. a software engineering discipline in which search-based optimization algorithms are used to address problems where a suitable balance between competing and potentially conflicting goals has to be found. Two key ingredients are required: (i) the representation of the problem; (ii) the definition of a fitness function. In fact, SBSE usually applies to problems in which there are numerous candidate solutions and where there is a fitness function that can guide the search process to locate reasonably good solutions.

A suitable representation of the problem allows to automatically explore the search space for the solutions that best fit the fitness function [72] that drives towards the sequence of the refactoring steps to apply to this system (i.e. altering its architectural structure without altering its semantics).

In the software performance domain both the suitable representation of the problem and the formulation of the fitness function are not trivial tasks, since the performance analysis results are derived from many uncertainties like the workload, the operational profile, etc. that might completely modify the perception of considering candidate solutions as good ones. Some assumptions can be introduced to simplify the problem and some design options can be explicitly defined in advance to constitute the *population* [72] on which search based optimization algorithms apply.

However, we believe that in the performance domain it is of crucial relevance to find a synergy between the search techniques that involve the definition of a fitness function to automatically capture what is required from the system, and the antipatterns that might support such function with the knowledge of bad practices and suggest common solutions, in order to quickly converge towards performance improvements.

In fact, as recently outlined in [71], there is a mutually beneficial relationship between SBSE and predictive models. In particular eleven broad areas of open problems (e.g. balancing functional, nonfunctional properties of predictive models) in SBSE for predictive modeling are discussed, explaining how techniques emerging from the SBSE community may find potentially innovative applications in predictive modeling.

2.3.1 Design Space Exploration Approaches

Zheng et al. in [140] described an approach to find optimal deployment and scheduling priorities for tasks in a class of distributed real-time systems. In particular, it is intended to evaluate the deployment of such tasks by applying a heuristic search strategy to LQN models. However, its scope is restricted to adjust the priorities of tasks competing for a processor, and the only refactoring action is to change the allocation of tasks to processors.

Bondarev et al. in [53] proposed a design space exploration methodology, i.e. DeSiX (DEsign, SImulate, eXplore), for software component-based systems. It adopts multi-dimensional quality attribute analysis and it is based on (i) various types of models for software components, processing nodes, memories and bus links, (ii) scenarios of system critical execution, allowing the designer to focus only on relevant static and dynamic system configurations, (iii) simulation of tasks automatically reconstructed for each scenario, and (iv) Pareto curves [54] for identification of optimal architecture alternatives.

An evolution of [53] can be found in [30], where a design space exploration framework for component-based software systems is presented. It allows an architect to get insight into a space of possible design alternatives with further evaluation and comparison of these alternatives. However, it requires a manual definition of design alternatives of software and hardware architectures, and it is meant to only identify bottlenecks.

Ipek et al. in [79, 80] described an approach to automatically explore the design space for hardware architectures, such as multiprocessors or memory hierarchies. The multiple design space points are simulated and the results are used to train a neural network. Such network can be solved quickly for different architecture candidates and delivers accurate results with a prediction error of less than 5%. However, the approach is limited to hardware properties, whereas software architectures are more complex, because architectural models spread on a wide rage of features.

2.3.2 Metaheuristic Approaches

Canfora et al. in [35] used genetic algorithms for Quality of Service (QoS)-aware service composition, i.e. to determine a set of *concrete* services to be bound to the *abstract* ones in the workflow of a composite service. However, each basic service is considered as a black-box element, where performance metrics are fixed to a certain unit (e.g. cost=5, resp.time=10), and the genetic algorithms search the best solutions by evaluating the composition options. Hence, no real feedback (in terms of refactoring actions in the software architectural model such as split a component) is given to the designer, with the exception of pre-defined basic services.

Aleti et al. in [16] presented a framework for the optimization of embedded system architectures. In particular, it uses the AADL (Architecture Analysis and Description Language) [57] as the underlying architecture description language and provides plug-in

mechanisms to replace the optimization engine, the quality evaluation algorithms and the constraints checking. Architectural models are optimized with evolutionary algorithms considering multiple arbitrary quality criteria. However, the only refactoring action the framework currently allows is the component re-deployment.

Martens et al. in [98] presented an approach for a performance-oriented design space exploration of component-based software architectures. An evolution of this work can be found in [99] where meta-heuristic search techniques are used for improving performance, reliability, and costs of of component-based software systems. In particular, evolutionary algorithms search the architectural design space for optimal trade-offs by means of Pareto curves. However, this approach is quite time-consuming, because it uses random changes (spanning on all feasible solutions) of the architecture, and the optimality is not guaranteed.

2.4 SUMMARY

Table 2.1 summarizes the main existing approaches in literature for the automated generation of architectural feedback. In particular, four categories of approaches are outlined: (i) antipattern-based approaches; (ii) rule-based approaches; (iii) design space exploration approaches; (iv) metaheuristic approaches.

The approach of this thesis somehow belongs to two categories, that are: antipattern-based and rule-based approaches. This is because it makes use of antipatterns for specifying rules that drive towards the identification of performance flaws.

Each *approach* is classified on the basis of the category it belongs to. Table 2.1 compares the different approaches by reporting the *(annotated) software architectural model* and the *performance model* they use to validate their applicability, if available. The last column of Table 2.1 denotes as *framework* the set of methodologies the corresponding approach entails. Note that in some cases the framework was implemented and it is available as a tool (e.g. SPE • ED, ArchE, PerOpteryx).

The framework we propose in this thesis, i.e. *PANDA* (*P*erformance *A*ntipatterns a*N*d Fee*D*back in Software *A*rchitectures), is meant to denote all the methodologies we propose in this research work and aimed at performing three main activities, i.e. specifying, detecting and solving antipatterns. Such framework is still a work in progress and we aim to implement it in the next future.

	Approach	(Annotated) Software Architectural Model	Performance Model	Framework
Antipatterns-based	Williams et al. [130], 2002	Software execution model (Execution graphs)	System execution model (Queueing Network)	SPE • ED
	Cortellessa et al. [43], 2007	Unified Modeling Language (UML)	Layered Queueing Network (LQN)	GARFIELD (Generator of Architectural Feedback through Performance Antipatterns Revealed)
	Parsons et al. [108], 2008	JEE systems from which component level end-to-end run-time paths are collected	Reconstructed run-time design model	PAD (Performance Antipattern Detection)
	This thesis, 2011	Unified Modeling Language (UML), Palladio Component Model (PCM)	Queueing Network, Simulation Model	PANDA (Performance Antipatterns aNd FeeDback in Software Architectures)
Rule-based	Barber et al. [21], 2002	Domain Reference Architecture (DRA)	Simulation Model	RARE and ARCADE
	Dobrzanski et al. [51], 2006	Unified Modeling Language (UML)	-	Telelogic TAU (i.e. UML CASE tool)
	McGregor et al. [101], 2007	Attribute-Driven Design (ADD)	Simulation Model	ArchE
	Kavimandan et al. [85], 2009	Real-Time Component Middleware	-	extension of the LwCCM middleware [106]
	Xu [138], 2010	Unified Modeling Language (UML)	Layered Queueing Network (LQN)	PB (Performance Booster)
Design Exploration	Zheng et al. [140], 2003	Unified Modeling Language (UML)	Simulation Model	-
	Bondarev et al. [30], 2007	Robocop Component Model	Simulation model	DeepCompass (Design Exploration and Evaluation of Performance for Component Assemblies)
	Ipek et al. [80], 2008	Artificial Neural Network (ANN)	Simulation Model	-
Metaheuristic	Canfora et al. [35], 2005	Workflow Model	Workflow QoS Model	-
	Aleti et al. [16], 2009	Architecture Analysis and Description Language (AADL)	Markov Model	ArcheOpterix
	Martens et al. [99], 2010	Palladio Component Model (PCM)	Simulation Model	PerOpteryx

Table 2.1: Summary of the approaches related to the generation of architectural feedback.

CHAPTER 3

PERFORMANCE ANTIPATTERNS

The goal of this Chapter is to describe performance antipatterns that, similarly to patterns, define recurring solutions to common design problems inducing performance flaws. An antipattern definition includes the problem (i.e. model properties that characterize the antipattern) and the solution (i.e. actions to take for removing the problem).

Up to now, in literature performance antipatterns have been only textually represented through informal language syntaxes [123]. The core question tackled in this Chapter is: how can a performance antipattern be represented to be automatically processed? To this aim, a notation-independent representation of performance antipatterns based on logical predicates is introduced.

3.1 INTRODUCTION

Figure 3.1 shows a first approach to the results interpretation & feedback generation steps of Figure 1.2. In order to make performance antipatterns machine-processable (that means detectable and solvable), we execute a preliminary *Modeling* step, that is matter of this Chapter and is represented in the rightmost rounded box of Figure 3.1, in which we specify antipatterns as logical predicates. Such predicates define conditions on architectural model elements (e.g. number of interactions among components, resource utilization, etc.), that allow to automate their detection. We have organized these architectural model elements in an *XML Schema* (see Appendix A).

Starting from an annotated software architectural model (label 5.a) and its performance indices (label 5.b), we execute an *Extracting* step during which the extractor engine generates an *XML representation of the Software System* conforming to our XML Schema and containing all and only the architectural model information we need to detect antipatterns.

The modeling of antipatterns entails the introduction of a set of *Boundaries* that drive the interpretation of performance analysis results, since they define thresholds that will be compared with the predicted values in order to decide the performance critical elements of the software architectural model (see more details in Section 3.4).

The *Detecting* step is the operational counterpart of the antipatterns declarative definitions as logical predicates. In fact, the detection engine takes as input the XML representation of the software system and the antipatterns boundaries, and it returns as output a list of *performance antipatterns instances*, i.e. the description of the detected problems as well as their solutions instantiated on the (annotated) software architectural model. Such list represents the feedback to the software designer, since it consists of a set of alternative refactoring actions, i.e. the backward path (label 6 of Figure 3.1), aimed at removing the detected antipatterns.

It is worth to notice that the formalization of performance antipatterns we propose is not heavily coupled with the detection engine: different types of engines can be implemented without modifying the formalization we define.

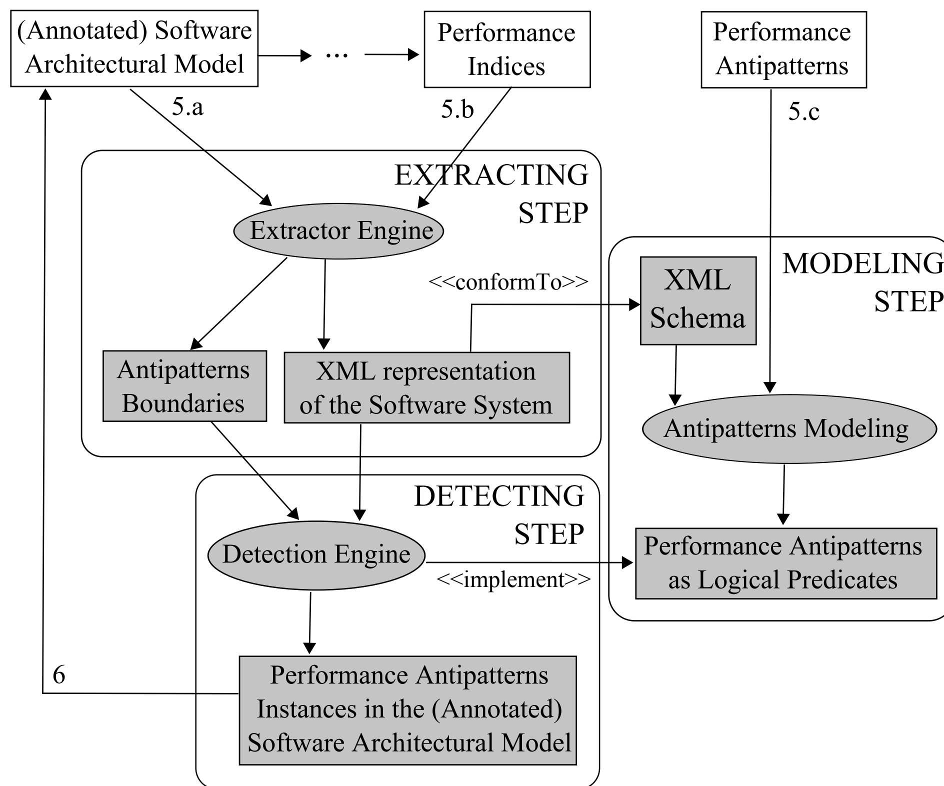


Figure 3.1: A first approach of the Results Interpretation & Feedback Generation steps.

This Chapter is organized as follows. Starting from the textual description of antipatterns (see Section 3.2), a graphical representation is provided to start reasoning on how antipattern specification may emerge from software architectural models (see Section 3.3). The logic-based formalization is explained in Section 3.4; it deals with the modeling of the antipattern *problem* specification, whereas some hints about the antipattern *solution* specification are given in Section 3.5.

3.2 Background: textual description

The main source of performance antipatterns is the work done across years by Smith and Williams [123] that have ultimately defined a number of 14 notation- and domain-independent antipatterns.

Table 3.1 lists the performance antipatterns we examine. Each row represents a specific antipattern that is characterized by three attributes: *antipattern* name, *problem* description, and *solution* description. From the original list [123], two antipatterns are not considered for the following reasons: the *Falling Dominoes* antipattern refers not only to performance problems, it includes also reliability and fault tolerance issues, therefore it is out of our interest; the *Unnecessary Processing* antipattern deals with the semantics of the processing by judging the importance of the application code, therefore it works at an abstraction level typically not observed in software architectural models.

The list of performance antipatterns defined by Smith and Williams has been here enriched with an additional attribute. As shown in the leftmost part of Table 3.1, we have partitioned antipatterns in two different categories.

The first category collects antipatterns detectable by single values of performance indices (such as mean, max or min values), and they are referred as *Single-value* Performance Antipatterns. The second category collects those antipatterns requiring the trend (or evolution) of the performance indices during the time (i.e. multiple values) to capture the performance problems induced in the software system. For these antipatterns, the mean, max or min values are not sufficient unless these values are referred to several observation time frames. They are referred as *Multiple-values* Performance Antipatterns and, due to these characteristics, the performance indices needed to detect such antipatterns must be obtained via system simulation or monitoring.

Some approaches [141, 139] have been recently introduced to monitor runtime data and update the performance model parameters. In [141] the track of runtime data is performed by means of Kalman Filters [32]. A Kalman Filter estimates model parameters in a recursive way, feeding back to the filter the difference between predictions and measurements, like response times and utilizations. The performance model dynamics are mathematically represented with a sensitivity matrix that configures the filter according to the behavior of the underlying selected performance model, typically a Queueing Network. As long as some easily verifiable conditions apply and enough measurements are provided in each step of the methodology, the estimations converge and increases the model parameter accuracy while reducing noise from systematic errors.

A similar reasoning is also followed in [56] where Bayesian Estimation techniques [23] are applied on the runtime data to update the model parameters of Discrete Time Markov Chains (DTMC) [95] or Queueing Networks [88]. However, particular focus is here put to reliability prediction.

In [139] a methodology for predicting the performance trends of a system with hidden pa-

	Antipattern		Problem	Solution
Single-value	Blob (or god class/component)		Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behavior together.
	Unbalanced Processing	Concurrent Processing Systems	Occurs when processing cannot make use of available processors.	Restructure software or change scheduling algorithms to enable concurrent execution.
		"Pipe and Filter" Architectures	Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput.	Break large filters into more stages and combine very small ones to reduce overhead.
		Extensive Processing	Occurs when extensive processing in general impedes overall response time.	Move extensive processing so that it does not impede high traffic or more important work.
	Circuitous Treasure Hunt		Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each look).
	Empty Semi Trucks		Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.	The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.
	Tower of Babel		Occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML.	The Fast Path performance pattern identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths by using the Coupling performance pattern to match the data format to the usage patterns.
	One-Lane Bridge		Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.	To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.
	Excessive Dynamic Allocation		Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.	1) Recycle objects (via an object pool) rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.
Multiple-values	Traffic Jam		Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.
	The Ramp		Occurs when processing time increases as the system is used.	Select algorithms or data structures based on maximum size or use algorithms that adapt to the size.
	More is Less		Occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources.	Quantify the thresholds where thrashing occurs (using models or measurements) and determine if the architecture can meet its performance goals while staying below the thresholds.

Table 3.1: Performance Antipatterns: problem and solution [123].

rameters is described. It combines a performance model with an autoregressive model to predict hidden parameters ahead in time, with estimation and prediction via an Extended Kalman Filter. Authors demonstrate that such combination greatly improves the prediction of trends. The integration of Extended Kalman filters to estimate model parameters appears as a promising approach to support the detection of Multiple-values performance antipatterns.

3.3 A GRAPHICAL REPRESENTATION

The aim of this Section is to provide a graphical representation of performance antipatterns on the basis of their textual description (see Table 3.1) in order to quickly convey the basic concepts of antipatterns. The graphical representation reflects our interpretation of the textual description of performance antipatterns [123], here visualized in a UML-like notation for a quick comprehension.

The graphical representation we propose is conceived to capture one reasonable illustration of both the antipattern problem and solution, but it does not claim to be exhaustive. Either the problem or even more the solution description of antipatterns give rise to a set of options that can be further considered to improve the current interpretation[1].

Similarly to the Three-View Model that was introduced in [132] for the performance engineering of software systems, the graphical representation is aimed at highlighting Static, Dynamic, and Deployment features to depict the presence (i.e. antipattern problem) or the absence (i.e. antipattern solution) of a certain set of architectural models properties.

3.3.1 SINGLE-VALUE PERFORMANCE ANTIPATTERNS

In this Section we report the graphical representation of the *Performance Antipatterns* that can be detected by single values of performance indices (such as mean, max or min values).

BLOB (OR GOD CLASS)

Problem *- Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.*

The blob antipattern may occur in two different cases.

[1]Especially for the representation of the antipattern solution, many options can be devised for refactoring actions (see more details in Section 3.5).

In the first case a single class or component contains most part of the application logics while all the other classes or components are used as data containers that offer only accessor functions, i.e. typically *get()* and *set()* methods. The typical behavior of such kind of class or component, called *Blob-controller* in the following, is to get data from other classes or components, perform a computation and then update data on the other classes or components.

In the second case a single class or component is used to store most part of the data of the application and it performs no application logic. A large set of other classes or components perform all the computation by getting data from such kind of class or component, called *Blob-dataContainer* in the following, through its *get()* methods and by updating data through its *set()* methods.

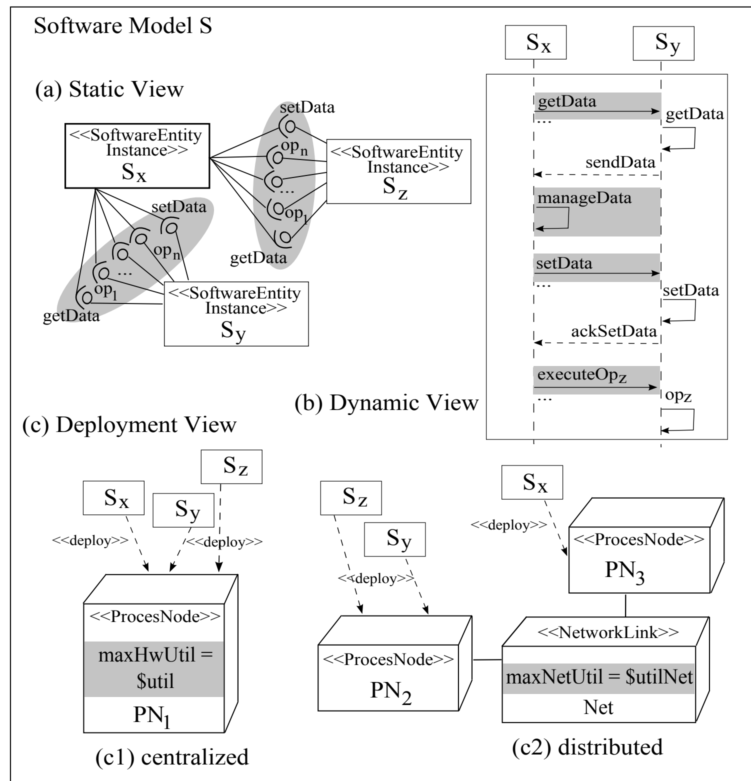
Both forms of the blob result from a poorly distributed system intelligence, i.e. a poor design that splits data from the relative processing logic. It might occurs in legacy systems, often composed by a centralized computing entity or by a unique data container, that are upgraded to object oriented ones without a proper re-engineering design analysis. The performance impacts of both cases are mainly due to the consequent excessive message passing among the blob software entity and the other classes or components. The performance loss is clearly heavier on distributed systems, where the time needed to pass data between remote software entities is significant with respect to the computational time.

Solution - *Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behavior together.*

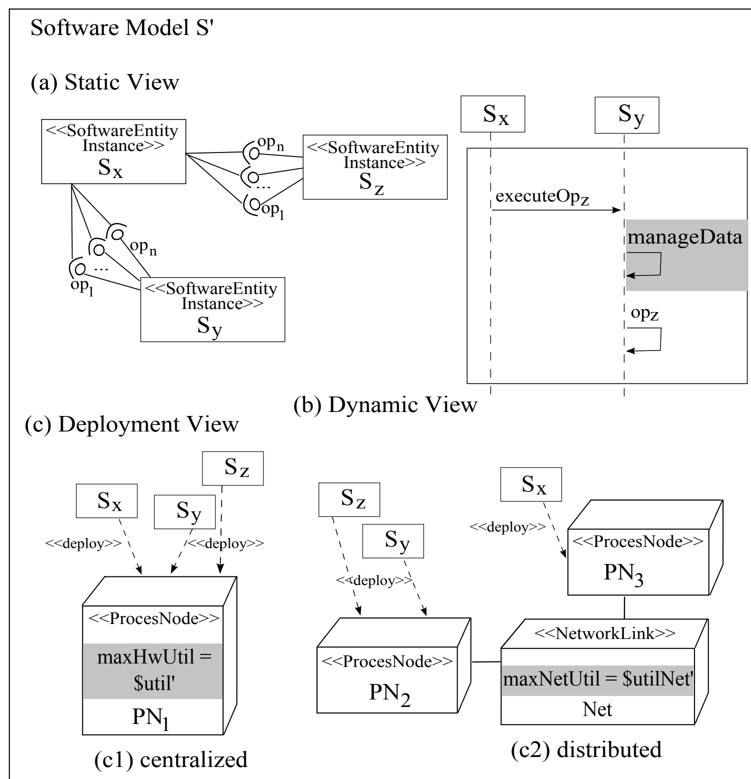
The solution to the blob antipattern is to refactor the design, because it is important to keep related data and behavior together. A software entity should keep most of the data that it needs to make a decision. The performance gain for the refactored solution will be $T_s = M_s \times O$, where T_s is the processing time saved, M_s is the number of messages saved and O is the overhead per message. The amount of overhead for a message will depend on the type of call, for example a local call will have less overhead than a remote procedure call.

Figures 3.2 and 3.3 provide a graphical representation of the *Blob* antipattern in its two forms, i.e. *Blob-controller* and *Blob-dataContainer* respectively.

The upper side of Figures 3.2 and 3.3 describes the properties of a *Software Model S* with a *BLOB problem*: (a) *Static View*, a complex software entity instance, i.e. S_x, is connected to other software instances, e.g. S_y and S_z, through *many* dependencies; (b) *Dynamic View*, the software instance S_x generates (see Figure 3.2) or receives (see Figure 3.3) *excessive* message traffic to elaborate data managed by other software instances such as S_y; (c) *Deployment View*, it includes two sub-cases: (c1) the centralized case, i.e. if the communicating software instances are deployed on the same processing node then a shared resource will show *high* utilization value, i.e. $\$util$; (c2) the distributed case, i.e. if the communicating software instances are deployed on different nodes then the

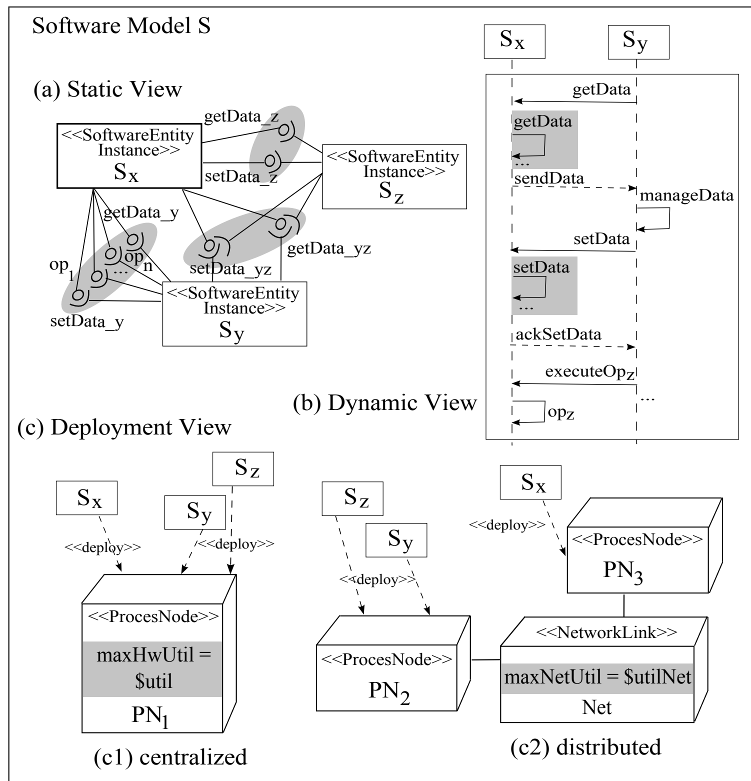


"BLOB-controller" problem

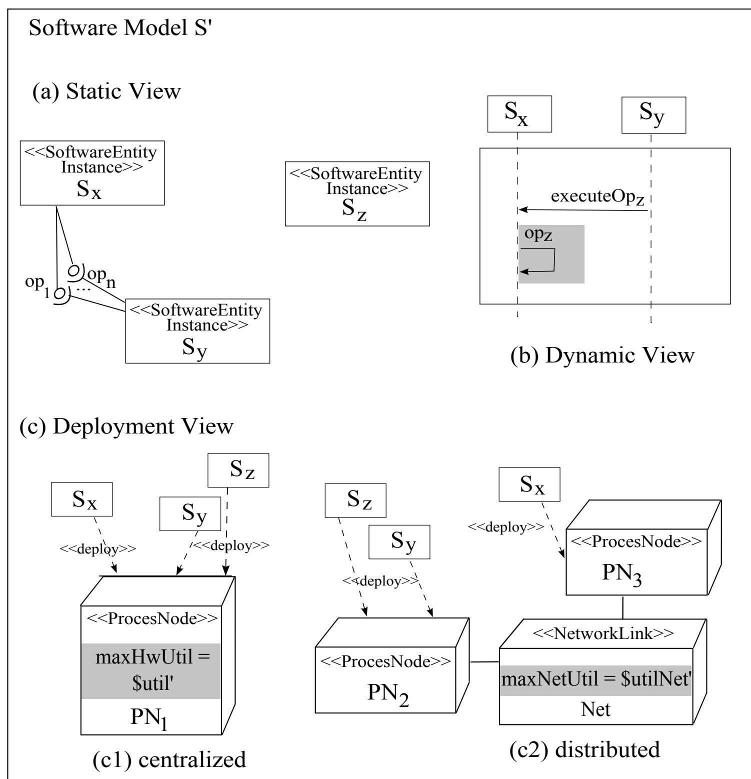


"BLOB-controller" solution

Figure 3.2: A graphical representation of the *Blob-controller* Antipattern.



"BLOB-dataContainer" problem



"BLOB-dataContainer" solution

Figure 3.3: A graphical representation of the *Blob-dataContainer* Antipattern.

network link will be a critical resource with a *high* utilization value, i.e. $\$utilNet$ [2]. The occurrence of such properties leads to assess that the software resource S_x originates an instance of the Blob antipattern.

The lower side of Figures 3.2 and 3.3 contains the design changes that can be applied according to the *BLOB solution*. The following refactoring actions are represented: (a) the number of dependencies between the software instance S_x and the surrounding ones, like S_y and S_z, must be decreased by delegating some functionalities to other instances; (b) the number of messages sent (see Figure 3.2) or received (see Figure 3.3) by S_x must be decreased by removing the management of data belonging to other software instances. As consequences of previous actions: (c1) if the communicating software instances were deployed on the same hardware resource then the latter will not be a critical resource anymore, i.e. $\$util' \ll \$util$; (c2) if the communicating software instances are deployed on different hardware resources then the network will not be a critical resource anymore, i.e. $\$utilNet' \ll \$utilNet$.

CONCURRENT PROCESSING SYSTEMS

Problem - *Occurs when processing cannot make use of available processors.*

The concurrent processing systems antipattern represents a manifestation of the unbalanced processing antipattern [121]. It occurs when processes cannot make effective use of available processors either because of 1) a non-balanced assignment of tasks to processors or because of 2) single-threaded code. In the following we only consider the case 1), since the application code is an abstraction level typically not included in architectural models.

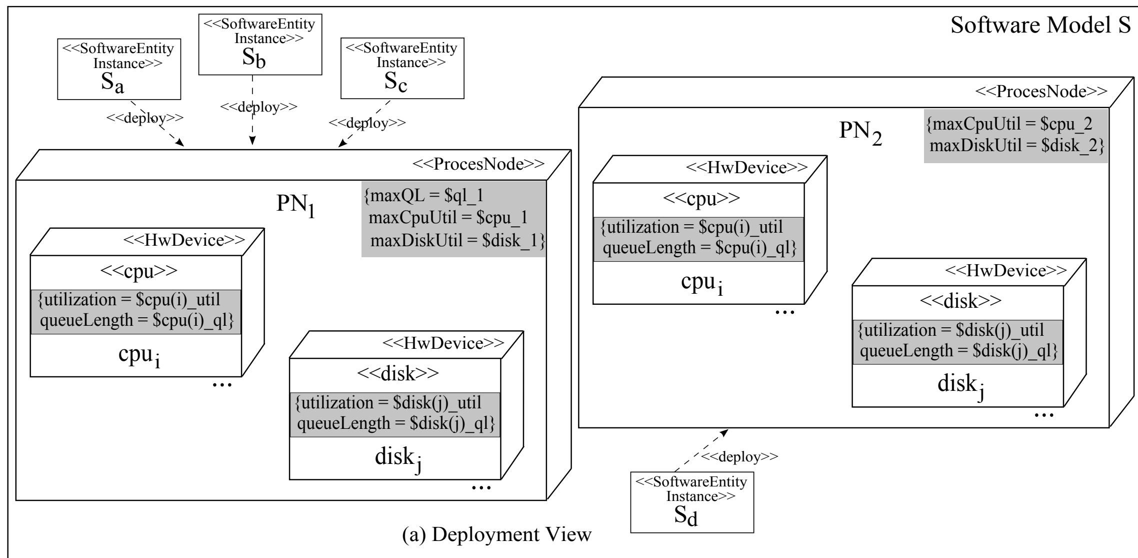
Solution - *Restructure software or change scheduling algorithms to enable concurrent execution.*

If a routing algorithm is based on static properties that result in more work going to one queue than others, than the solution is to use a dynamic algorithm that routes work to queues based on the work requirements and the system congestion.

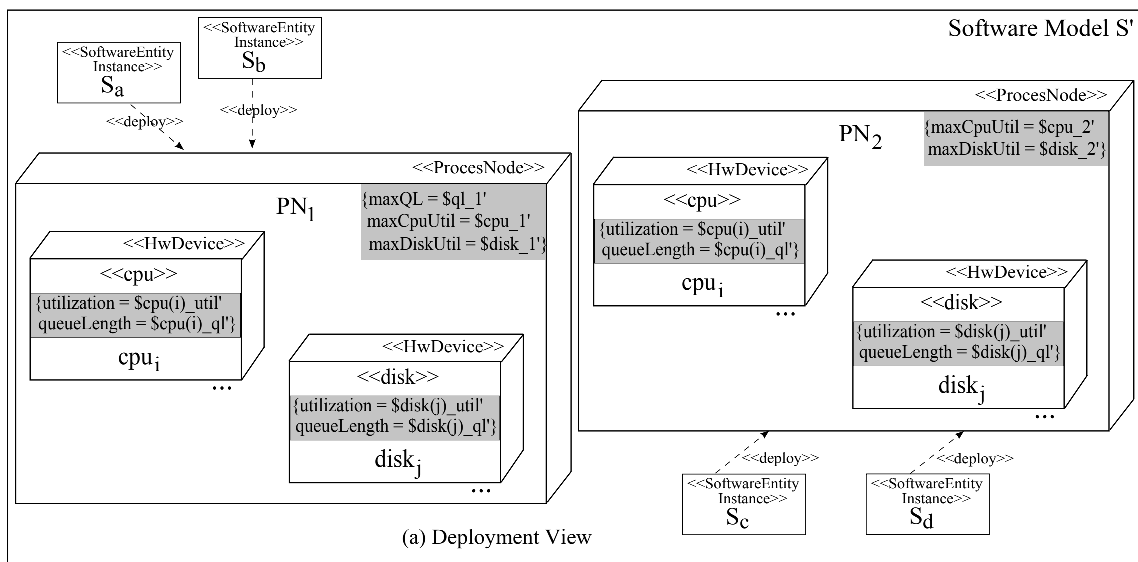
Figure 3.4 provides a graphical representation of the *Concurrent Processing Systems* antipattern.

The upper side of Figure 3.4 describes the system properties of a *Software Model S* with a *Concurrent Processing Systems problem*: (a) *Deployment View*, there are two processing nodes, e.g. PN_1 and PN_2, with un unbalanced processing, i.e. many tasks (e.g. computation from the software entity instances S_a, S_b, S_c) are assigned to PN_1 whereas PN_2 is not so heavily used (e.g. computation of the software entity instance S_d). The

[2]The characterization of antipattern parameters related to system characteristics (e.g. *many* usage dependencies, *excessive* message traffic) or to performance results (e.g. *high*, *low* utilization) is based on thresholds values (see more details in Section 3.4).



"Concurrent Processing Systems" problem



"Concurrent Processing Systems" solution

Figure 3.4: A graphical representation of the *Concurrent Processing Systems* Antipattern.

over used processing node will show *high* queue length value, i.e. $\$ql_1$ (estimated as the maximum value overall its hardware devices, i.e. $\$cpu(i)_ql$ and $\$disk(j)_ql$), and a *high* utilization value among its hardware entities either for cpus, i.e. $\$cpu_1$ (estimated as the maximum value overall its cpu devices, i.e. $\$cpu(i)_util$), and disks, i.e. $\$disk_1$, devices (estimated as the maximum value overall its disk devices, i.e. $\$disk(j)_util$). The less used processing node will show *low* utilization value among its hardware entities either for cpus, i.e. $\$cpu_2$, and disks, i.e. $\$disk_2$, devices. The occurrence of such properties leads to assess that the processing nodes PN_1 and PN_2 originate an instance of the Concurrent Processing Systems antipattern.

The lower side of Figure 3.4 contains the design changes that can be applied according to the *Concurrent Processing Systems solution*. The following refactoring actions are represented: (a) the software entity instances must be deployed in a better way, according to the available processing nodes. As consequences of the previous action, if the software instances are deployed in a balanced way then the processing node PN_1 will not be a critical resource anymore, hence $\$ql_1'$, $\$cpu_1'$, $\$disk_1'$ values improves despite the $\$cpu_2'$, $\$disk_2'$ values.

PIPE AND FILTER ARCHITECTURES

Problem *- Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput.*

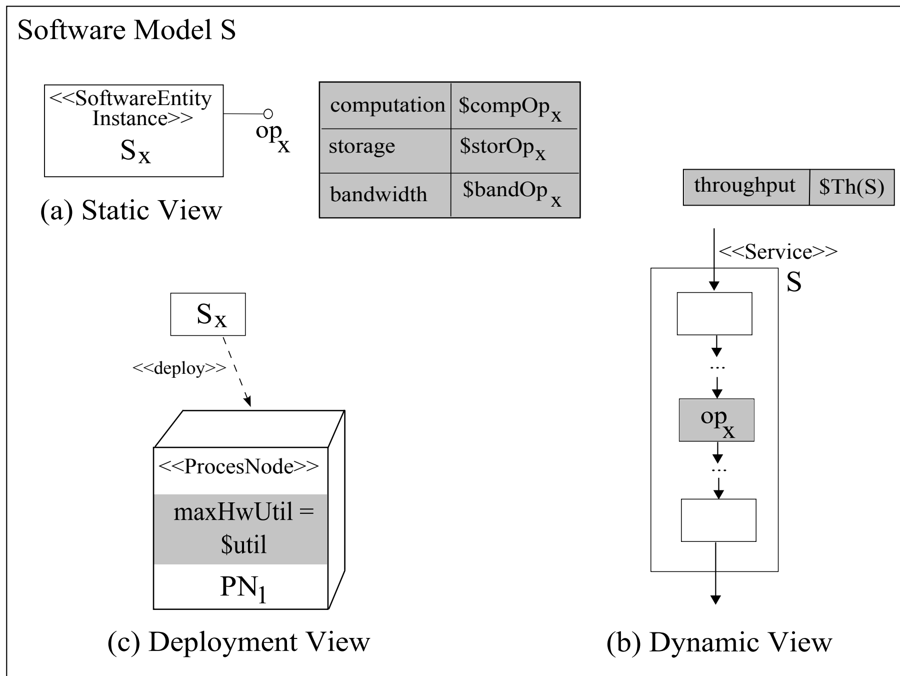
The pipe and filter architectures antipattern represents a manifestation of the unbalanced processing antipattern [121]. It occurs when the throughput of the overall system is determined by the slowest filter. It means that there is a stage in a pipeline which is significantly slower than all the others, therefore constituting a bottleneck in the whole process in which most stages have to wait the slowest one to terminate.

Solution *- Break large filters into more stages and combine very small ones to reduce overhead.*

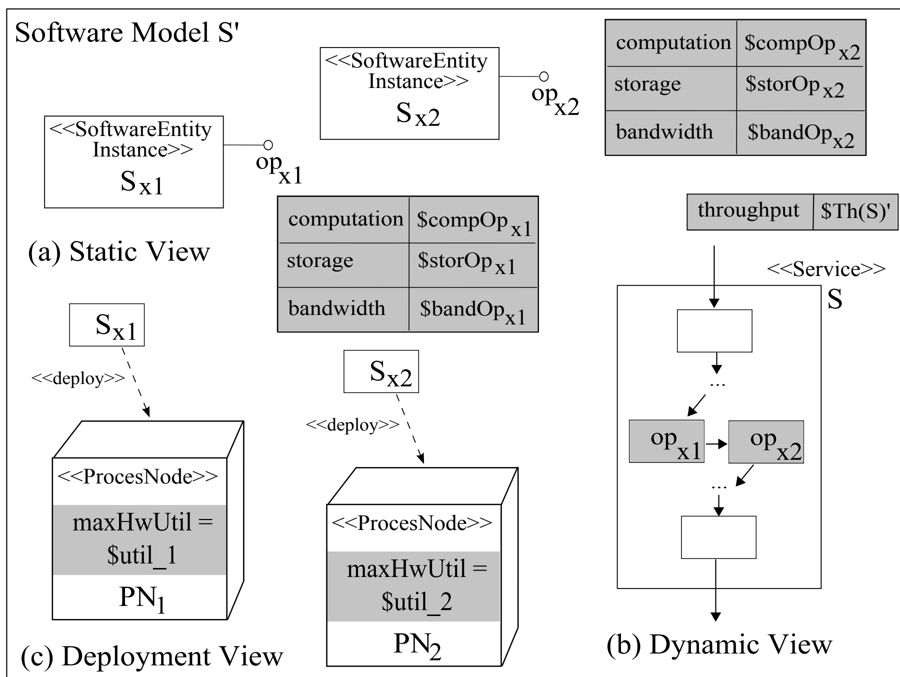
The solution to the pipe and filter architectures antipattern is to 1) divide long processing steps into multiple, smaller stages that can execute in parallel; 2) combine short processing steps to minimize context switching overhead and other delays for shared resources.

Figure 3.5 provides a graphical representation of the *Pipe and Filter Architectures* antipattern.

The upper side of Figure 3.5 describes the system properties of a *Software Model S* with a *Pipe and Filter Architectures problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, offering an operation (op_x) whose resource demand (computation = $\$compOp_x$, storage = $\$storOp_x$, bandwidth = $\$bandOp_x$) is quite *high*; (b) *Dynamic View*, the operation op_x is invoked in a service S and the throughput of the service ($\$Th(S)$) is lower than the required one; (c) *Deployment View*, the processing node on which S_x is deployed, i.e.



"Pipe and Filter Architectures" problem



"Pipe and Filter Architectures" solution

Figure 3.5: A graphical representation of the *Pipe and Filter Architectures* Antipattern.

PN_1, might have a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the operation op_x originates an instance of the Pipe and Filter Architectures antipattern.

The lower side of Figure 3.5 contains the design changes that can be applied according to the *Pipe and Filter Architectures solution*. The following refactoring actions are represented: (a) the operation op_x must be divided in at least two operations, i.e. op_{x1} and op_{x2}, offered by two different software instances; (b) software instances deployed on different processing nodes enable the parallel execution of requests. As consequences of the previous actions, if the operations are executed in parallel then the operation op_x will not be the slowest filter anymore. The throughput of the service S is expected to improve, i.e. $\$Th(S)' > \$Th(S)$.

EXTENSIVE PROCESSING

Problem - *Occurs when extensive processing in general impedes overall response time.*

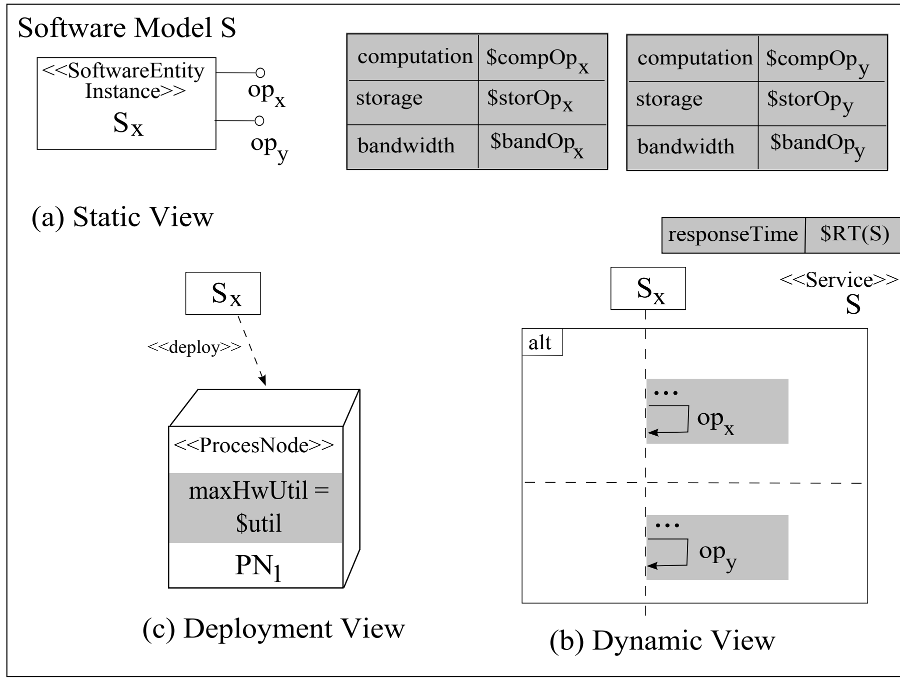
The extensive processing antipattern represents a manifestation of the unbalanced processing antipattern [121]. It occurs when a long running process monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation. The processor is removed from the pool, but unlike the pipe and filter, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload.

Solution - *Move extensive processing so that it does not impede high traffic or more important work.*

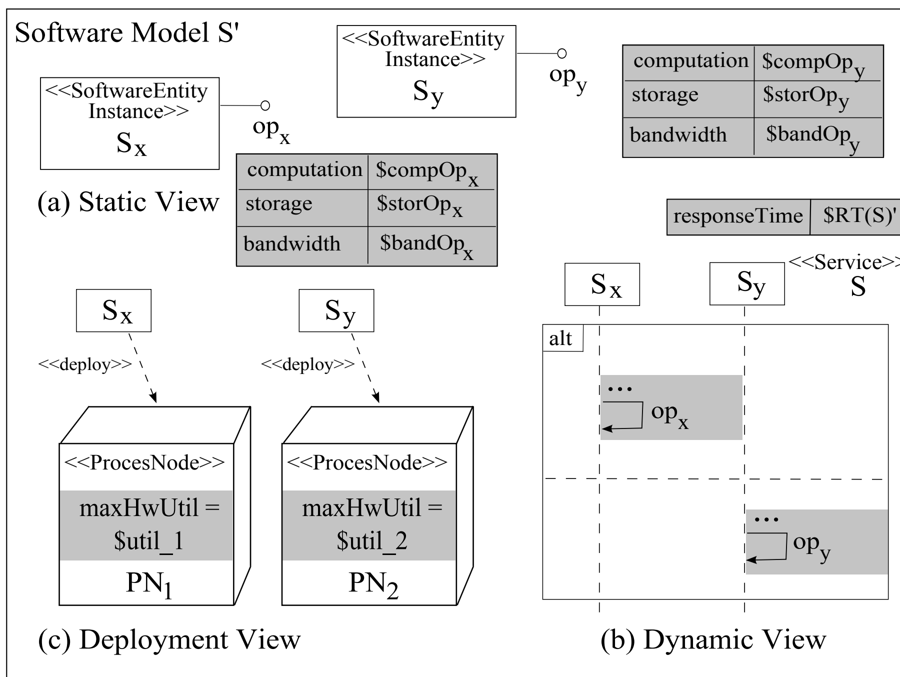
The solution to the extensive processing antipattern is to identify processing steps that may cause slow downs and delegate those steps to processes that will not impede the fast path. A performance improvement could be achieved by delegating processing steps which do not need a synchronous execution to other processes.

Figure 3.6 provides a graphical representation of the *Extensive Processing* antipattern.

The upper side of Figure 3.6 describes the system properties of a *Software Model S* with a *Extensive Processing problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, offering two operations (op_x, op_y) whose resource demand is quite unbalanced, since op_x has a *high* demand (computation = $\$compOp_x$, storage = $\$storOp_x$, bandwidth = $\$bandOp_x$), whereas op_y has a *low* demand (computation = $\$compOp_y$, storage = $\$storOp_y$, bandwidth = $\$bandOp_y$); (b) *Dynamic View*, the operations op_x and op_y are alternatively invoked in a service S, and the response time of the service ($\$RT(S)$) is larger than the required one; (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might reveal a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the operations op_x and op_y originate an instance of the



"Extensive Processing" problem



"Extensive Processing" solution

Figure 3.6: A graphical representation of the *Extensive Processing* Antipattern.

Extensive Processing antipattern.

The lower side of Figure 3.6 contains the design changes that can be applied according to the *Extensive Processing solution*. The following refactoring actions are represented: (a) the operations op_x and op_y must be offered by two different software instances; (b) software instances deployed on different processing nodes provide a fast path for requests. As consequences of the previous actions, the response time of the service S is expected to improve, i.e. $\$RT(S)' < \$RT(S)$.

CIRCUITOUS TREASURE HUNT

Problem - *Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer.*

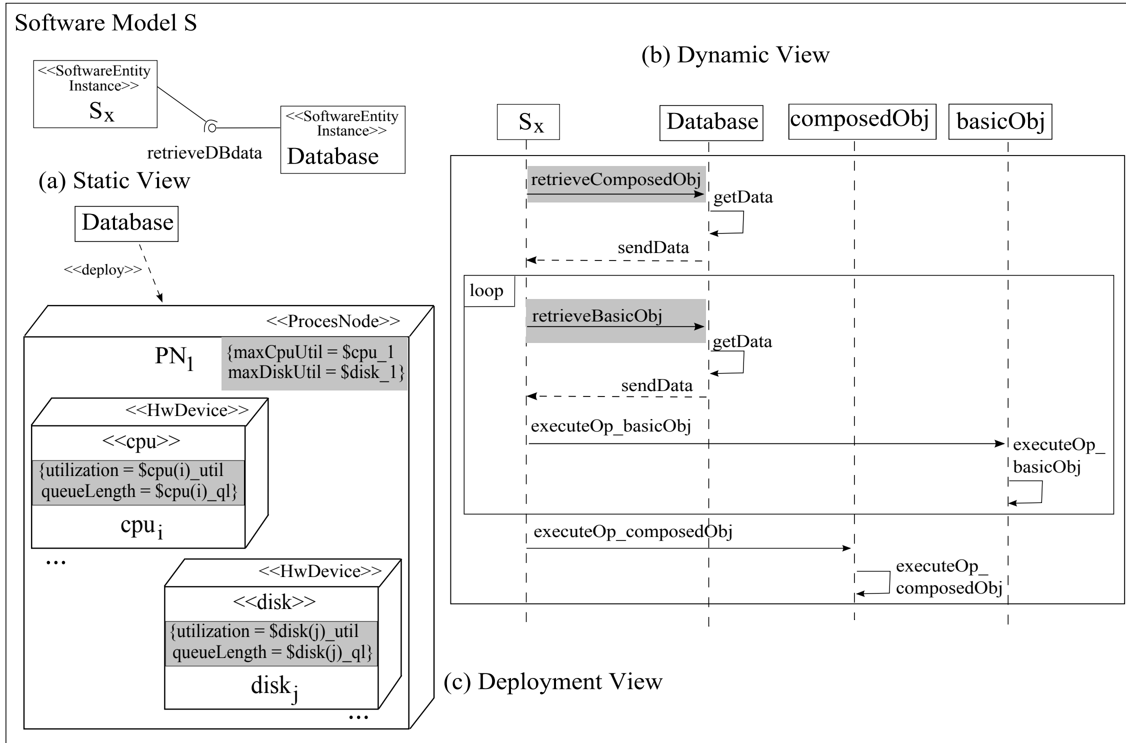
The circuitous treasure hunt antipattern typically occurs in database applications when the computation necessitates a high amount of database requests to retrieve the data. It happens that a software instance performs a sequence of queries to obtain some data by exploiting the information retrieved in a query to construct the next one, instead of properly constructing a single, even more complex, query that in a faster way achieve the needed information. The performance loss is due to the overhead introduced by the cost of database access, query processing and the transmission of all intermediate results. The problem is particularly heavy in distributed systems. Object oriented systems are prone to this antipattern because it might happen that a chain of method calls among objects is performed to execute an operation or retrieve some data: an object invokes a method of another object which calls a third one in a chain which ends when the final operation is performed or when the data is found. The performance loss is due to the extra processing required to identify the operation to be performed on each object of the chain, to the number of request messages and to the overhead needed to send requested data among all the objects of the chain.

Solution - *Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each look).*

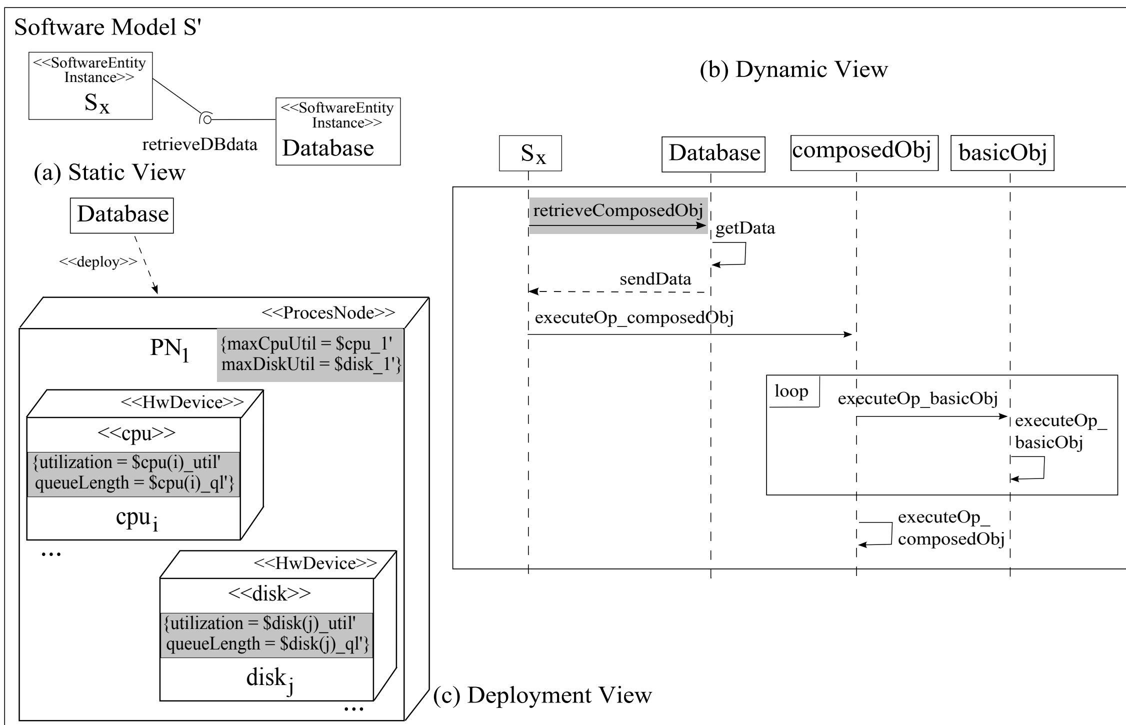
The solution to the circuitous treasure hunt antipattern is to better design the queries performed to a database or, in some cases, to redesign data organization in the database. Another solution is to reduce the performance loss by introducing an adapter entity, which consists in the creation of a single software instance handling part of the logics needed to perform a query and providing an interface to other objects in order to reduce the traffic between the objects and the database query handler.

Figure 3.7 provides a graphical representation of the *Circuitous Treasure Hunt* antipattern.

The upper side of Figure 3.7 describes the system properties of a *Software Model S* with a *Circuitous Treasure Hunt problem*: (a) *Static View*, there is a software entity instance,



"Circuitous Treasure Hunt" problem



"Circuitous Treasure Hunt" solution

Figure 3.7: A graphical representation of the *Circuitous Treasure Hunt* Antipattern.

e.g. S_x, retrieving some information from the database; (b) *Dynamic View*, the software instance S_x generates a *large* number of database calls by performing several queries up to the final operation; (c) *Deployment View*, the processing node on which the database is deployed, i.e. PN_1, might have a *high* utilization value among its devices ($\$cpu_1$, $\$disk_1$). Furthermore, a database transaction usually requires an higher utilization of disk devices instead of cpu ones, hence $\$disk_1$ is expected to be larger than $\$cpu_1$. The occurrence of such properties leads to assess that the software instance $Database$ originates an instance of the Circuitous Treasure Hunt antipattern.

The lower side of Figure 3.7 contains the design changes that can be applied according to the *Circuitous Treasure Hunt solution*. The following refactoring action is represented: (a) the database must be restructured to reduce the number of database calls and to retrieve the needed information with a fewer number of database transactions. As consequences of the previous action, if the information is retrieved with a smarter organization of the database than the utilization of hardware devices is expected to improve in the *Software Model S'*, i.e. $\$cpu_1'$ and particularly $\$disk_1'$.

EMPTY SEMI TRUCKS

Problem - *Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.*

The problem of inefficient use of available bandwidth typically affects message-based systems when a huge load of messages, each containing a small amount of information, is exchanged over the network. The amount of processing overhead is the same regardless of the size of the message. With smaller messages, this processing is required many more times than necessary hence it significantly implies a performance loss. The problem of an inefficient interface (i.e. it provides a too fragmented access to data) generates an excessive overhead caused by the computation needed to handle each call request.

Solution - *The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.*

In the case of inefficient use of available bandwidth the solution is given by the adoption of *Batching* performance pattern [122] that basically groups information in larger chunks in order to minimize the overhead due to information spread and processing. In this way several messages are merged into a single bigger message. The time savings, T, is $T = (t_p + t_r) \times M$ where t_p is the time for preparation, i.e. the processing time for acquiring message buffers as well as transmission overhead, sending message headers, etc., t_r is the time for the receipt, i.e. the similar processing time and transmission overhead for acknowledgements, etc., M is the number of messages eliminated. In the case of inefficient interface a solution could be achieved through the implementation of the *Coupling* performance pattern [122] that basically uses more coarse-grained objects

in order to reduce the amount of communication overhead required to obtain data.

Figure 3.8 provides a graphical representation of the *Empty Semi Trucks* antipattern.

The upper side of Figure 3.8 describes the system properties of a *Software Model S* with a *Empty Semi Trucks problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, retrieving some information from several instances ($Srem_1$, ..., $Srem_n$); (b) *Dynamic View*, the software instance S_x generates an *excessive* message traffic by sending a big amount of messages by *low* sizes ($\$msgS$), much lower than the network bandwidth, hence the network link might have a *low* utilization value ($\$utilNet$); (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might reveal a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the software instance S_x originates an instance of the Empty Semi Trucks antipattern.

The lower side of Figure 3.8 contains the design changes that can be applied according to the *Empty Semi Trucks solution*. The following refactoring action is represented: (a) the communication between S_x and the remote instances must be restructured, messages are merged in bigger ones ($\$msgS'$) to reduce the number of messages sent over the network. As consequences of the previous action, if the information is exchanged with a smarter organization of the communication than the utilization of the processing node hosting S_x is expected to improve, i.e. $\$util' \ll \$util$.

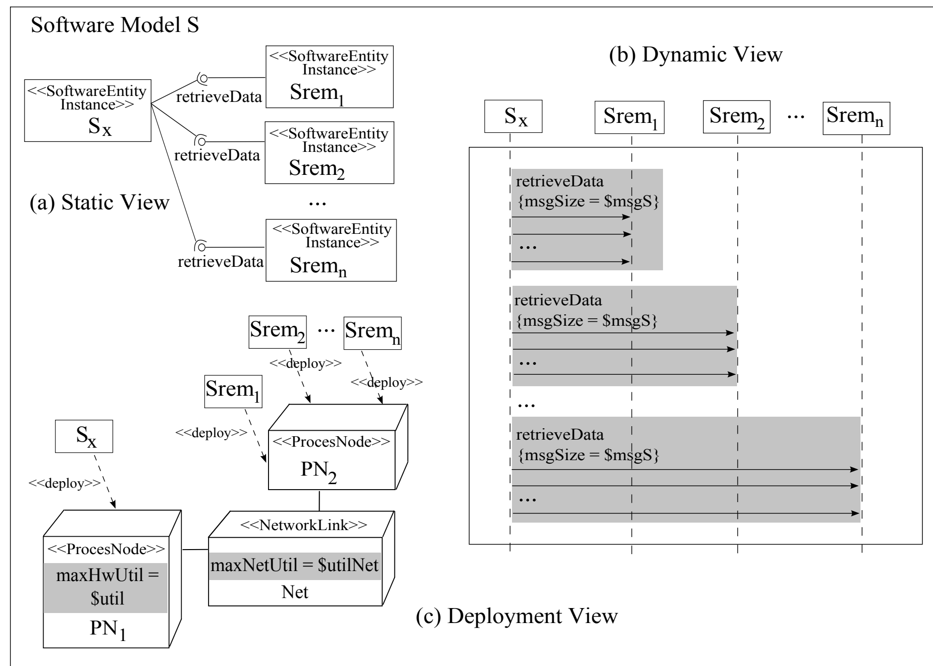
TOWER OF BABEL

Problem *- Occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML.*

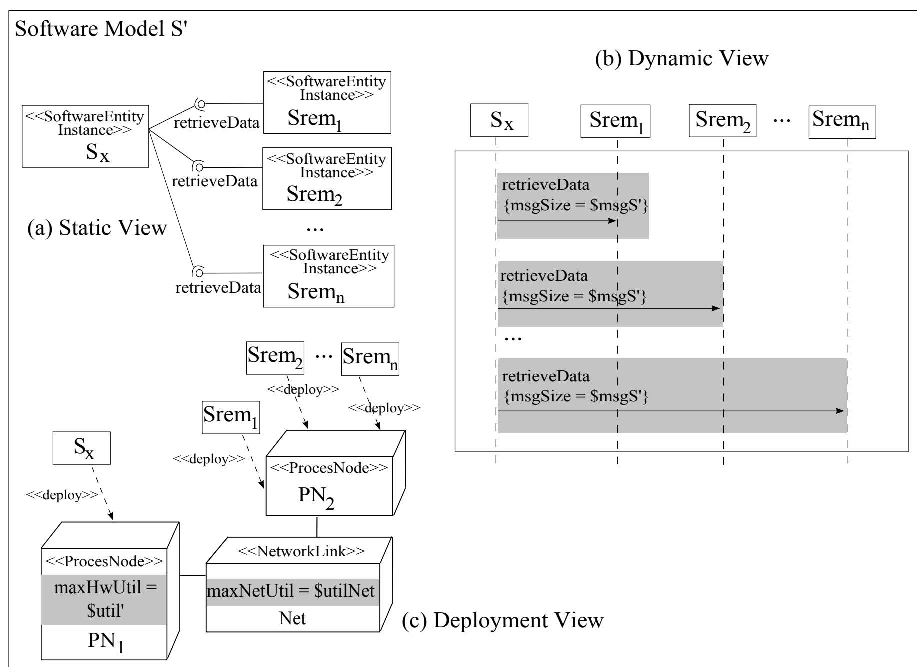
The antipattern occurs in complex, distributed data-oriented systems in which the same information is often translated into an exchange format (by a sending process) and then parsed and translated into an internal format (by the receiving process). The problem is that when the translation and parsing is excessive the system spends most of its time doing this and relatively little doing real work. The performance loss in this case is clearly due to the excessive overhead caused by the translation and parsing operations which may be executed several times in the whole execution process.

Solution *- The Fast Path performance pattern identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths by using the Coupling performance pattern to match the data format to the usage patterns.*

A performance improvement can be achieved by deciding a common format which minimizes the operations of translation along the core, fastest path identified through the *Fast Path* performance pattern [122]. In simpler cases, a good solution could be easily found by avoiding unnecessary translations among software entities, typically introduced to adopt standard exchange languages, even when not necessary. The time savings, T, is $T = (s_c + s_p + s_t) \times 2N$ where s_c is the service time to convert the internal format



"Empty Semi Trucks" problem



"Empty Semi Trucks" solution

Figure 3.8: A graphical representation of the *Empty Semi Trucks* Antipattern.

to the intermediate format to send the request or response, s_p is the service time to parse the intermediate format, s_t is the service time to translate the intermediate format into the internal format, N is the number of processes in the end-to-end scenario that require the translation to/from the intermediate format. The multiplier of $2N$ in the formula is because the entire process of converting, parsing and translating apply to both the input and the reply messages.

Figure 3.9 provides a graphical representation of the *Tower of Babel* antipattern.

The upper side of Figure 3.9 describes the system properties of a *Software Model S* with a *Tower of Babel problem*: (a) *Static View*, there are some software entity instances, e.g. S_x, S_1, ..., S_n; (b) *Dynamic View*, the software instance S_x performs *many* times the translation of format for communicating with other instances; (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might reveal a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the software instance S_x originates an instance of the Tower of Babel antipattern.

The lower side of Figure 3.9 contains the design changes that can be applied according to the *Tower of Babel solution*. The following refactoring action is represented: (a) the communication between S_x and the other instances can be restructured by setting the format. As consequence of the previous action the utilization of the processing node hosting S_x is expected to improve, i.e. $\$util' \ll \$util$.

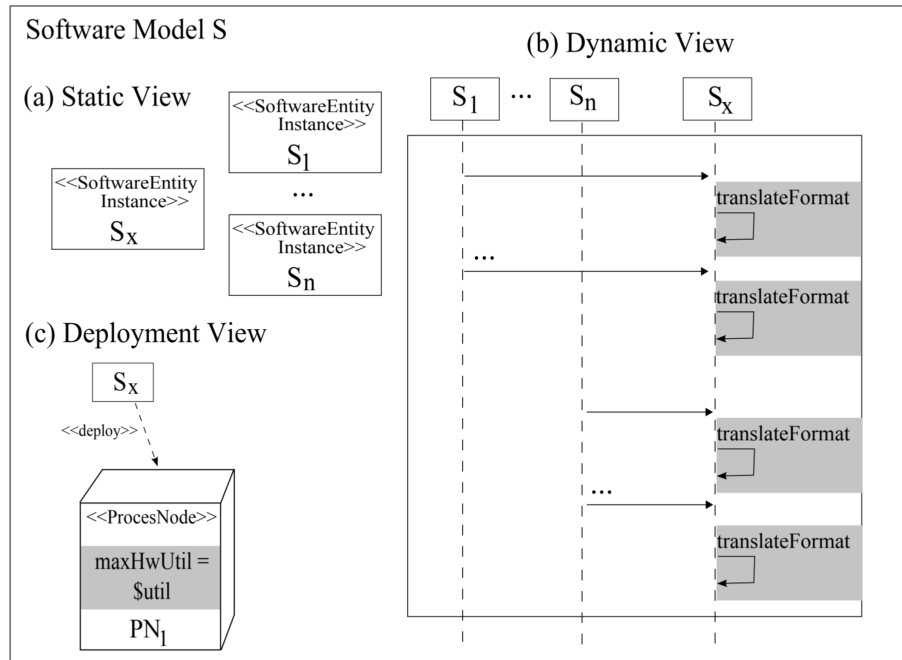
ONE-LANE BRIDGE

Problem - *Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.*

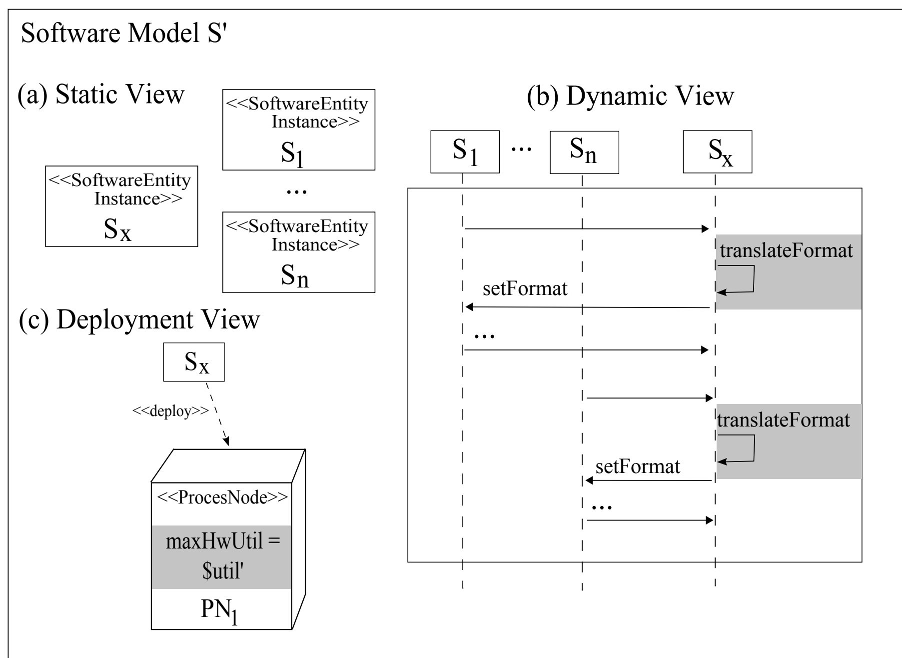
The one-lane bridge antipattern often occurs in concurrent systems when the mechanisms of mutual access to a shared resource are badly designed. The problem appears in all cases of mutual access to resources, or in the case of multiple processes making synchronous calls in a non multi-threaded environment. In that case a single process is able to handle a single method invocation at a time, thus forcing all the other processes calling it to wait for the end of the execution of the previous call. It frequently occurs also in database applications. A lock ensures that only one process can update the associated portion of the database at a time, hence when a process is updating data it keeps the lock over a portion of the data set, and other processes must stop and wait for the release of the lock in order to access the needed information.

Solution - *To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.*

In case of mutual exclusion and of synchronous computation in single-threaded architectures the solution can be re-engineer the shared data and the access to them by respect-



"Tower of Babel" problem



"Tower of Babel" solution

Figure 3.9: A graphical representation of the *Tower of Babel* Antipattern.

ing the principles of mutual access, or by the adoption of a multi-threaded architecture. In case of database applications the solution could be found by splitting data sets into smaller ones, i.e. by distributing information contained in a single, big table, into a set of smaller tables. Other solutions can be re-engineer the information structure in the database accordingly to the most performed operation of update. The general principle is to share resources when possible, and if exclusive access is required, the holding and the scheduling times should be minimized.

Figure 3.10 provides a graphical representation of the *One-Lane Bridge* antipattern.

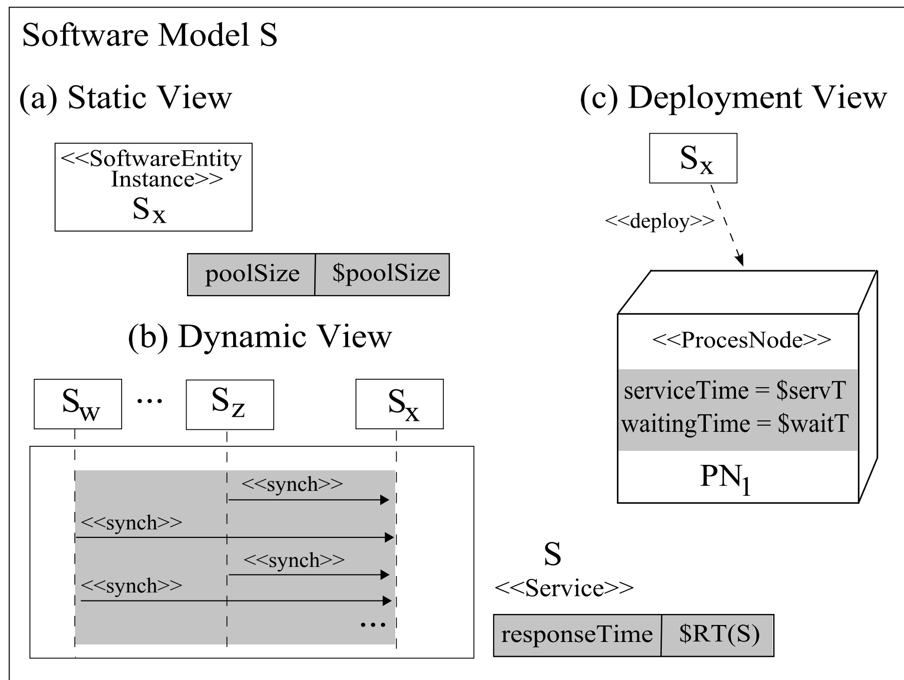
The upper side of Figure 3.10 describes the system properties of a *Software Model S* with a *One-Lane Bridge problem*: (a) *Static View*, there is a software entity instance with a capacity of managing $\$poolSize$ threads; (b) *Dynamic View*, the software instance S_x receives an *excessive* number of synchronous calls in a service S and the predicted response time, i.e. $\$RT(S)$, is higher than the required one; (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, might reveal *high* service and waiting times, i.e. $\$servT$ and $\$waitT$. The occurrence of such properties leads to assess that the software instance S_x originates an instance of the One-Lane Bridge antipattern.

The lower side of Figure 3.10 contains the design changes that can be applied according to the *One-Lane Bridge solution*. The following refactoring action is represented: (a) the pool size of the software instance S_x must be increased. As consequences of the previous action, if the software entity is able to cope with synchronous calls than the response time of the service S is expected to improve, i.e. $\$RT(S)' < \$RT(S)$, as well as the service and waiting times of the processing node hosting S_x, i.e. $\$servT'$ and $\$waitT'$.

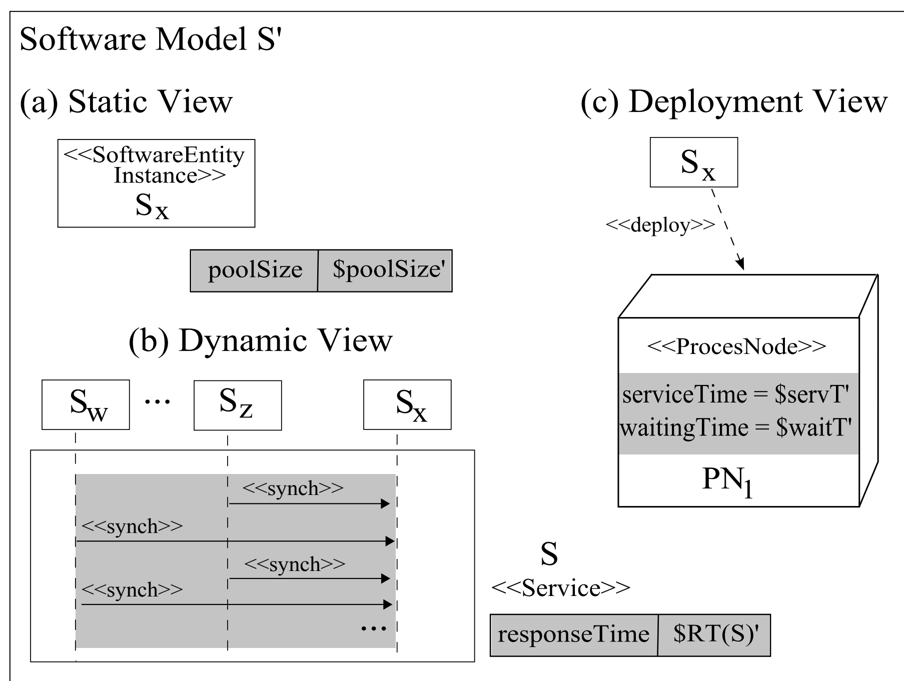
EXCESSIVE DYNAMIC ALLOCATION

Problem *- Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.*

Dynamic allocation is an useful technique used in dynamic systems in order to avoid excessive memory storage when objects are rarely used: an object is created when needed and immediately destroyed afterwards. Sometimes however this approach is over-adopted even with objects that are often used by other objects. In that case the memory storage benefit is minimized, while a significant computation overhead is introduced due to the cost of instantiation of an object, which needs the memory allocation on the heap, the initialization computation logic, and the deletion and clean-up operations when the object is destroyed. In distributed systems, the performance loss is even larger due to the overhead for the creation and the deletion of exchange messages. The cost of dynamic allocation is $C = N \times \sum_{depth}(s_c + s_d)$ where N is the number of calls, $depth$ is the number of contained objects that must be created when the class is created, and s_c and s_d are the times to create and to destroy the object, respectively.



"One-Lane Bridge" problem



"One-Lane Bridge" solution

Figure 3.10: A graphical representation of the *One-Lane Bridge* Antipattern.

Solution *- 1) Recycle objects (via an object pool) rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.*

Two solutions are devised for the excessive dynamic allocation antipattern. The first solution consists in recycling objects rather than creating new ones each time they are needed. This approach pre-allocates a pool of objects and stores them in a collection. New instances of the object are requested from the pool, and unneeded instances are returned to it. This approach is useful for systems that continually need many short-lived objects. There is a pre-allocation of objects at the system initialization but the run-time overhead is reduced to simply passing a pointer to the pre-allocated object. The second solution consists in sharing the same object among all the objects that need to access it.

Figure 3.11 provides a graphical representation of the *Excessive Dynamic Allocation* antipattern.

The upper side of Figure 3.11 describes the system properties of a *Software Model S* with an *Excessive Dynamic Allocation problem*: (a) *Static View*, there is a software entity instance S_x offering an operation op_x; (b) *Dynamic View*, the software instance S_x creates and destroys a *high* number of objects for performing the operation op_x in a service S and the predicted response time, i.e. $\$RT(S)$, is higher than the required one. The occurrence of such properties leads to assess that the software instance S_x originates an instance of the Excessive Dynamic Allocation antipattern.

The lower side of Figure 3.11 contains the design changes that can be applied according to the *Excessive Dynamic Allocation solution*. The following refactoring action is represented: (a) the object used for performing the operation op_x is created and destroyed once. As consequences of the previous action, the response time of the service S is expected to improve, i.e. $\$RT(S)' < \$RT(S)$.

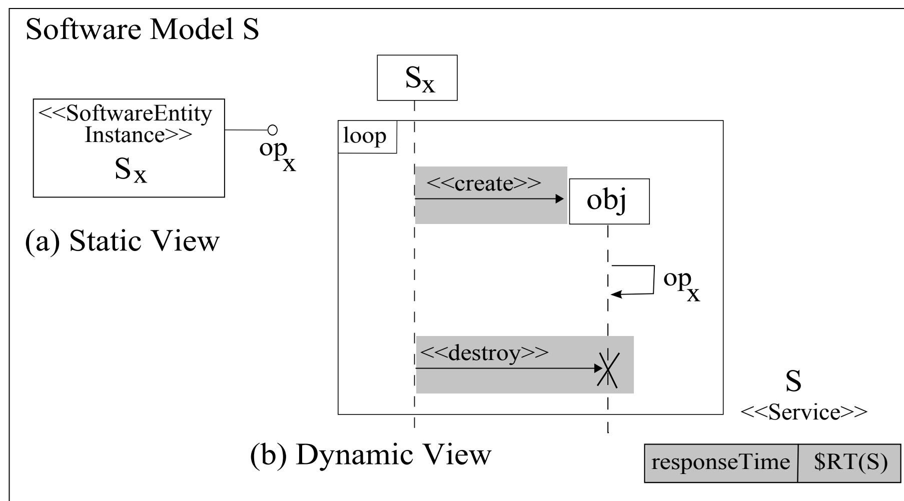
3.3.2 MULTIPLE-VALUES PERFORMANCE ANTIPATTERNS

In this Section we report the graphical representation of the *Performance Antipatterns* that can be detected only observing the trend (or evolution) of the performance indices along the time (i.e., multiple-values).

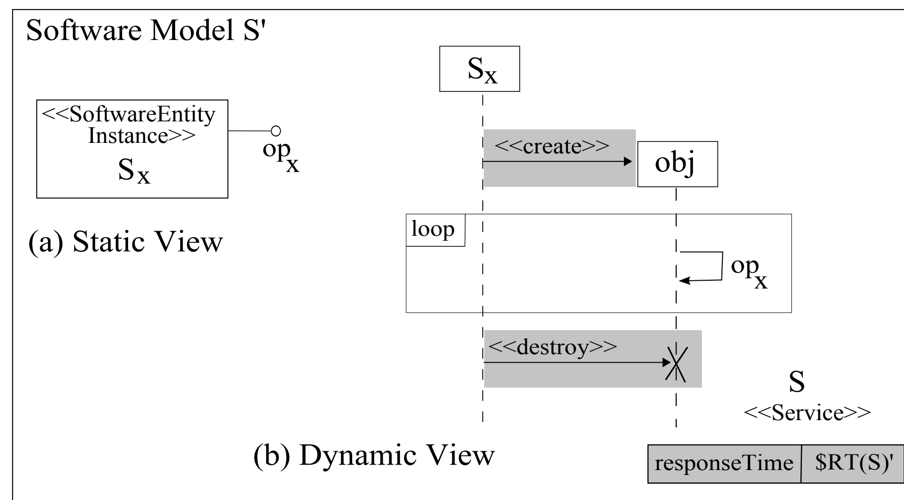
TRAFFIC JAM

Problem *- Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.*

The traffic jam antipattern occurs if a significative variability in response time is observed, and it is due to different causes, usually difficult to identify. The problem also occurs when a large amount of work is scheduled within a relatively small interval. It occurs, for example, when a huge number of processes are originated at approximately the same



"Excessive Dynamic Allocation" problem



"Excessive Dynamic Allocation" solution

Figure 3.11: A graphical representation of the *Excessive Dynamic Allocation* Antipattern.

time. The challenge is that often the performance loss and the slowing down of the computation arises and persists for a long time after the originating cause has disappeared. A failure or any other cause of performance bottleneck generate a backlog of jobs waiting for computation, which take a long time to be executed and to return in a normal operating condition.

Solution *- Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.*

If the problem is caused by periodic high demand, it might be beneficial to seek alternatives that spread the load, or handle the demand in a different manner. For example, if users select the time for performing a request to the system, it may be helpful to change the selection options so that they select a time interval rather than selecting a specific time. It gives the software more flexibility in scheduling the requests in order to reduce contention. If the problem is caused by external factors, such as domain specific behaviors, than it is important to determine the size of the platforms and networks that will be required to support the worst-case workload intensity. The ideal solution is clearly given by the elimination of the originating bottleneck. However, localizing the source of the backlog is not simple and the increase of the computing power, even costly, could benefit.

Figure 3.12 provides a graphical representation of the *Traffic Jam* antipattern.

The upper side of Figure 3.12 describes the system properties of a *Software Model S* with a *Traffic Jam problem*: (a) *Static View*, there is a software entity instance S_x offering an operation op_x; the monitored response time of the operation op_x shows "*a wide variability in response time which persists long* " [120]. The occurrence of such properties leads to assess that the software instance S_x originates an instance of the Traffic Jam antipattern.

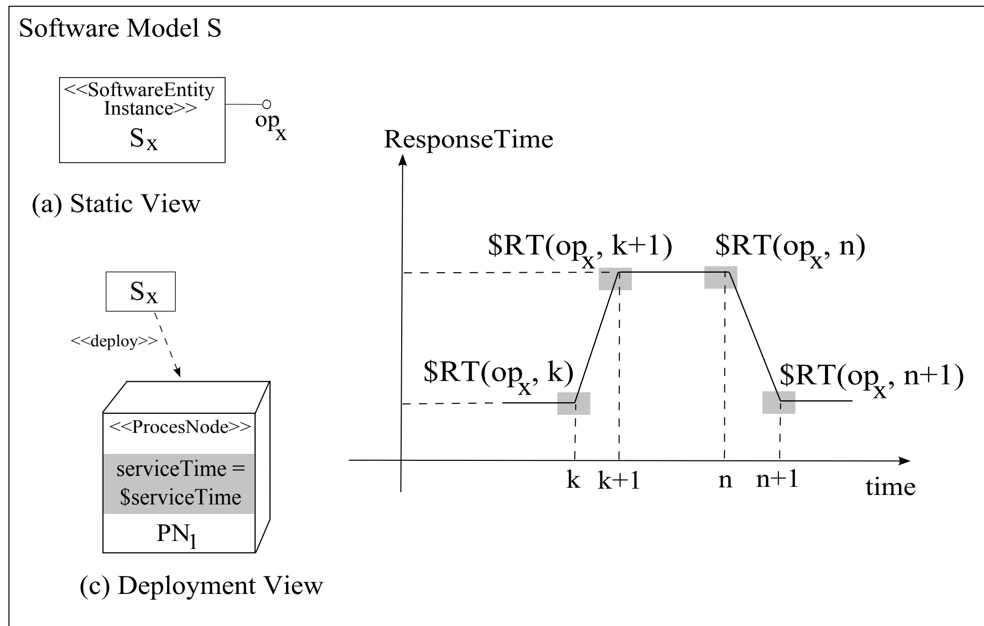
The lower side of Figure 3.12 contains the design changes that can be applied according to the *Traffic Jam solution*. The following refactoring action is represented: (a) the service time of the processing node on which S_x is deployed must be decreased, i.e. $\$serviceTime'$. As consequences of the previous action, the response time of the operation op_x is expected to increase in a slower way.

THE RAMP

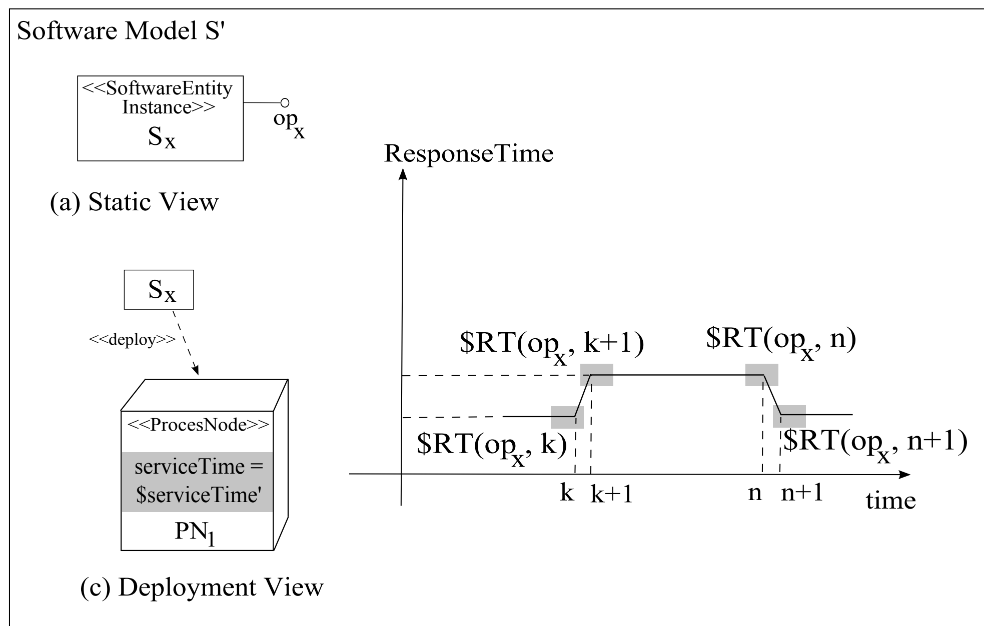
Problem *- Occurs when processing time increases as the system is used.*

The Ramp is an antipattern that addresses the situation where the amount of processing required by a system to satisfy a request increases over time. It is due to the growing amount of data the system stores and, as the time goes on, the data grow and the processing time required to perform an operation on such data becomes unacceptable. It is presented as a scalability problem, it is often not detected during testing since test data often does not contain enough items to reveal the phenomenon.

Solution *- Select algorithms or data structures based on maximum size or use algorithms*



"Traffic Jam" problem



"Traffic Jam" solution

Figure 3.12: A graphical representation of the *Traffic Jam* Antipattern.

that adapt to the size.

The solution is basically to keep the processing time from increasing dramatically as the data set grows. Possible solutions include: (i) select a search algorithm that is appropriate for a larger amount of data, it may be suboptimal for small sizes, but it should not hurt to do extra work then; (ii) automatically invoke self-adapting algorithms based on size; (iii) when the size increases more gradually, use instrumentation to monitor the size and upgrade the algorithm at predetermined points. The ideal solution is given by increasing the computing power because, even costly, it could benefit.

Figure 3.13 provides a graphical representation of *The Ramp* antipattern.

The upper side of Figure 3.13 describes the system properties of a *Software Model S* with *The Ramp problem*: (a) *Static View*, there is a software entity instance S_x offering an operation op_x; (b) *Dynamic View*: (i) the monitored response time of the operation op_x at time t_1, i.e. $\$RT(op_x, t_1)$, is *much* lower than the monitored response time of the operation op_x at time t_2, i.e. $\$RT(op_x, t_2)$, with $t_1 < t_2$; (ii) the monitored throughput of the operation op_x at time t_1, i.e. $\$Th(op_x, t_1)$, is *much* larger than the monitored throughput of the operation op_x at time t_2, i.e. $\$Th(op_x, t_2)$, with $t_1 < t_2$. The occurrence of such properties leads to assess that the software instance S_x originates an instance of The Ramp antipattern.

The lower side of Figure 3.13 contains the design changes that can be applied according to the *The Ramp solution*. The following refactoring action is represented: (a) the service time of the processing node on which S_x is deployed must be decreased, i.e. $\$serviceTime'$. As consequences of the previous action, the response time of the operation op_x is expected to increase in a slower way and the throughput of the operation op_x is expected to decrease in a slower way.

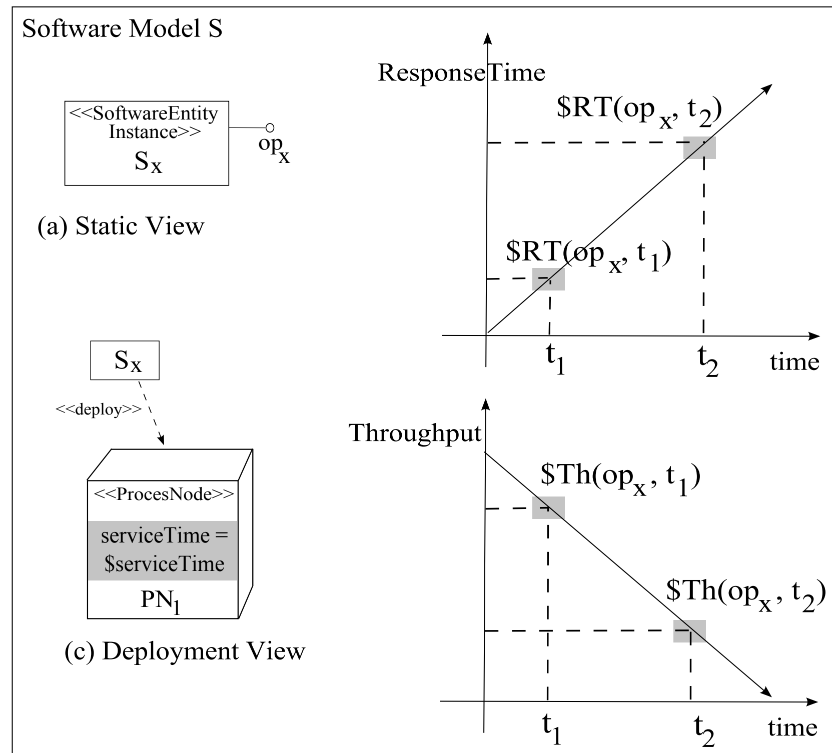
MORE IS LESS

Problem - *Occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources.*

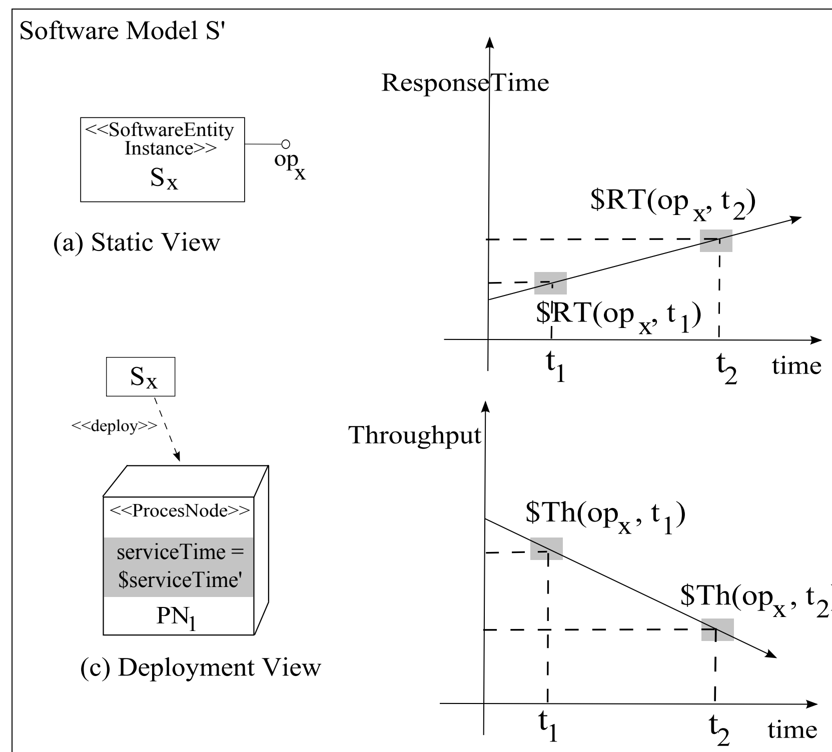
The antipattern occurs when running too many processes over time causes too much paging and too much overhead for servicing page faults. The same problem takes place in database systems when too many database connections are created by causing a significant performance loss. Also in distributed systems the same behavior could happen when too many internet connections or too many pooled resources are allowed, therefore there are too many concurrent streams relative to the number of available processors.

Solution - *Quantify the thresholds where thrashing occurs (using models or measurements) and determine if the architecture can meet its performance goals while staying below the thresholds.*

Computer systems have diminishing returns due to contention for resources. Therefore,



"The Ramp" problem

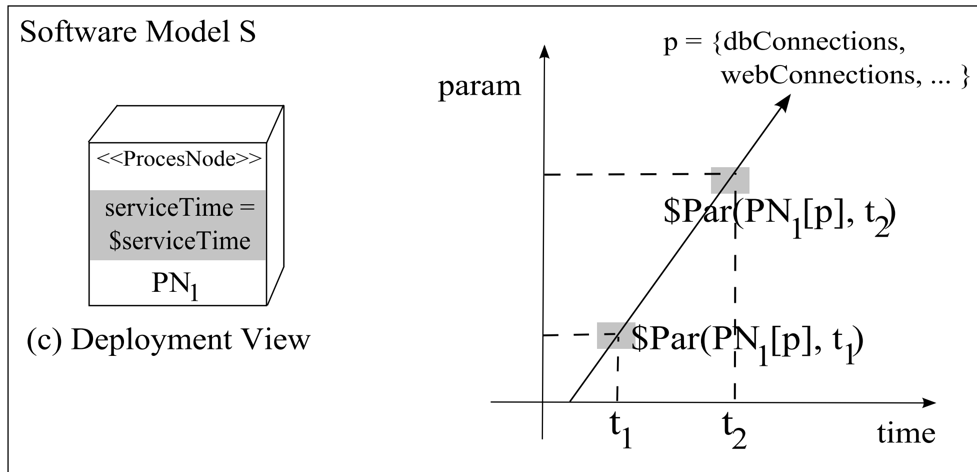


"The Ramp" solution

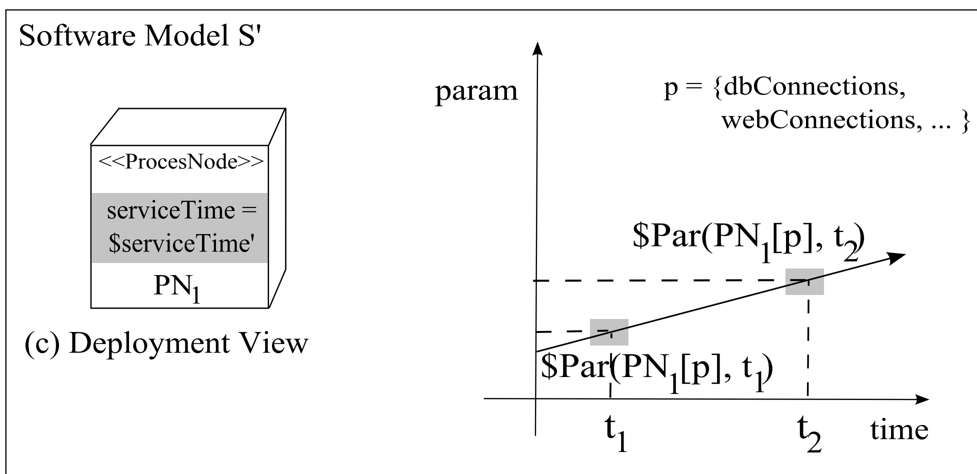
Figure 3.13: A graphical representation of *The Ramp* Antipattern.

the solution is to quantify the point when resource contention exceeds an acceptable threshold. Models or measurement experiments can identify these points. The ideal solution is given by increasing the computing power because, even costly, it could benefit.

Figure 3.14 provides a graphical representation of the *More Is Less* antipattern.



"More is Less" problem



"More is Less" solution

Figure 3.14: A graphical representation of the *More Is Less* Antipattern.

The upper side of Figure 3.14 describes the system properties of a *Software Model S* with a *More Is Less problem*: (a) *Deployment View*, there is a processing node PN_1 and the monitored runtime parameters (e.g. database connections, pooled resources, internet connections, etc.) at time t_1, i.e. $\$Par(PN_1[p], t_1)$, are *much* larger than the same parameters at time t_2, i.e. $\$Par(PN_1[p], t_2)$, with $t_1 < t_2$. The occurrence of such properties leads to assess that PN_1 originates an instance of the antipattern.

The lower side of Figure 3.14 contains the design changes that can be applied according

to the *More Is Less solution*. The following refactoring action is represented: (a) the service time of the processing node PN_1 must be decreased, i.e. $\$serviceTime'$. As consequences of the previous action, the monitored runtime parameters of the processing node PN_1 are expected to increase in a slower way.

3.4 A LOGIC-BASED SPECIFICATION OF THE ANTIPATTERN PROBLEM

In this Section performance antipatterns problems are formally defined as logical predicates. Such predicates define conditions on specific architectural model elements (e.g. number of interactions among software resources, hardware resources throughput) that we have organized in an XML Schema (see Appendix A).

The specification of *model elements* to describe antipatterns is a quite complex task, because such elements can be of different types: (i) elements of a software architectural model (e.g. software resource, message, hardware resource); (ii) performance results (e.g. utilization of a network resource); (iii) structured information that can be obtained by processing the previous ones (e.g. the number of messages sent by a software resource towards another one); (iv) bounds that give guidelines for the interpretation of the system features (e.g. the upper bound for the network utilization).

These two latter model elements, i.e. structured information and bounds, have been defined respectively by introducing supporting *functions* that elaborate a certain set of system elements (represented in the predicates as $F_{funcName}$), and *thresholds* that need to be compared with the observed properties of the software system (represented in the predicates as $Th_{thresholdName}$).

Thresholds must be bound to concrete numerical values, e.g. hardware resources whose utilization is higher than 0.8 can be considered critical ones. The binding of thresholds is a critical point of the whole approach, since they introduce uncertainty and must be suitably tuned. Some sources can be used to perform this task such as: (i) the system requirements; (ii) the domain expert's knowledge; (iii) the estimation of the system under analysis[3]. Both functions and thresholds introduced for representing antipatterns as logical predicates are summarized in Section 3.4.3.

One sub-section is dedicated to each antipattern and is organized as follows. From the informal representation of the *problem* (as reported in Table 3.1), a set of *basic predicates* (BP_i) is built, where each BP_i addresses part of the antipattern problem specification. The basic predicates are first described in a semi-formal natural language and then formalized by means of first-order logics. Note that the operands of basic predicates are elements of our XML Schema, here denoted with the `typewriter font`.

[3]For more details about the estimation of thresholds please refer to Tables 3.3 and 3.4.

Figure 3.15 provides a bird's eye look of our XML Schema (it is fully described in Appendix A) that sketches the three views and their intersection in the central and shaded square (i.e. views overlapping). A *Service* is defined at an higher level, because it can be described by means of elements belonging to all the three views.

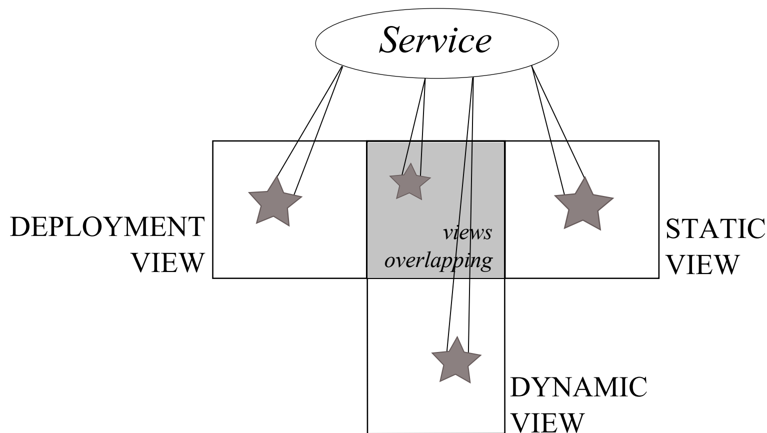


Figure 3.15: Bird's-eye look of the XML Schema and its views.

The benefit of introducing (static, dynamic, deployment) views is that the performance antipattern specification can be partitioned on their basis: the predicate expressing a performance antipattern is in fact the conjunction of sub-predicates, each referring to a different view. However, to specify an antipattern it might not be necessary information coming from all views, because certain antipatterns involve only elements of some views.

Note that in this process we provide our formal interpretation of the informal definitions of Table 3.1. Hence, the formalization we propose obviously reflects our interpretation. Different formalizations of antipatterns can be originated by laying on different interpretations of their informal definitions.

3.4.1 Single-value Performance Antipatterns

In this Section we report the *Performance Antipatterns* that can be detected by single values of performance indices (such as mean, max or min values).

Blob (or god class/component)

Problem Blob (or "god" class/component) [119] has the following problem informal definition: *"occurs when a single class either 1) performs all of the work of an application or 2) holds all of the application's data. Excessive message traffic that can degrade performance"* (see Table 3.1).

We formalize this sentence with four basic predicates: the BP_1 predicate whose elements belong to the Static View; the BP_2 predicate whose elements belong to the Dynamic

View; and finally the BP_3 and BP_4 predicates whose elements belong to Deployment View.

-STATIC VIEW-

BP_1- Two cases can be identified for the occurrence of the blob antipattern.

In the first case there is at least one `SoftwareEntityInstance`, e.g. swE_x, such that it "*performs all of the work of an application*" [119], while relegating other instances to minor and supporting roles. Let us define the function $F_{numClientConnects}$ that counts how many times the software entity instance swE_x is in a `Relationship` with other software entity instances by assuming a `clientRole` for swE_x. The property of performing all the work of an application can be checked by comparing the output value of the $F_{numClientConnects}$ function with a threshold $Th_{maxConnects}$:

$$F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \tag{3.1}$$

In the second case there is at least one `SoftwareEntityInstance`, e.g. swE_x, such that it "***holds all of the application's data***" [119]. Let us define the function $F_{numSupplierConnects}$ that counts how many times the software entity instance swE_x is in a `Relationship` with other software entity instances by assuming the `supplierRole` for swE_x. The property of holding all of the application's data can be checked by comparing the output value of the $F_{numSupplierConnects}$ function with a threshold $Th_{maxConnects}$:

$$F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects} \tag{3.2}$$

-DYNAMIC VIEW-

BP_2 - swE_x performs most of the business logics in the system or holds all the application's data, thus it generates or receives excessive message traffic. Let us define by $F_{numMsgs}$ the function that takes in input a software entity instance with a `senderRole`, a software entity instance with a `receiverRole`, and a `Service` S, and returns the multiplicity of the exchanged `Messages`. The property of excessive message traffic can be checked by comparing the output value of the $F_{numMsgs}$ function with a threshold $Th_{maxMsgs}$ in both directions:

$$F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \tag{3.3a}$$
$$F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs} \tag{3.3b}$$

-DEPLOYMENT VIEW-

The performance impact of the excessive message traffic can be captured by considering two cases. The first case is the *centralized* one (modeled by the BP_3 predicate), i.e.

the blob software entity instance and the surrounding ones are deployed on the same processing node, hence the performance issues due to the excessive load may come out by evaluating the utilization of such processing node. The second case is the *distributed* one (modeled by the BP_4 predicate), i.e. the Blob software entity instance and the surrounding ones are deployed on different processing nodes, hence the performance issues due to the excessive message traffic may come out by evaluating the utilization of the network links.

BP_3- The `ProcesNode` P_{xy} on which the software entity instances swE_x and swE_y are deployed shows heavy computation. That is, the `utilization` of a hardware entity of the `ProcesNode` P_{xy} exceeds a certain threshold $Th_{maxHwUtil}$. For the formalization of this characteristic, we use the $F_{maxHwUtil}$ function that has two input parameters: the processing node, and the type of `HardwareEntity`, i.e. 'cpu', 'disk', or 'all' to denote no distinction between them. In this case the $F_{maxHwUtil}$ function is used to determine the maximum `Utilization` among 'all' the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \tag{3.4}$$

BP_4- The `ProcesNode` P_{swE_x} on which the software entity instance swE_x is deployed, shows a high utilization of the network connection towards the `ProcesNode` P_{swE_y} on which the software entity instance swE_y is deployed. Let us define by $F_{maxNetUtil}$ the function that provides the maximum value of the `usedBandwidth` overall the network links joining the processing nodes P_{swE_x} and P_{swE_y}. We must check if such value is higher than a threshold $Th_{maxNetUtil}$:

$$F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil} \tag{3.5}$$

Summarizing, the *Blob (or "god" class/component)* antipattern occurs when the following composed predicate is true:

$$\exists swE_x, swE_y \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid$$

$$\boxed{((3.1) \vee (3.2)) \wedge ((3.3a) \vee (3.3b)) \wedge ((3.4) \vee (3.5))}$$

where $\mathrm{sw}\mathbb{E}$ represents the `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, swE_y, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Blob antipattern.

CONCURRENT PROCESSING SYSTEMS

Problem Concurrent Processing Systems [121] has the following problem informal definition: *"occurs when processing cannot make use of available processors"* (see Table 3.1).

We formalize this sentence with three basic predicates: the BP_1, BP_2, BP_3 predicates whose elements belong to the Deployment View. In the following, we denote with $\mathbb{P}$ the set of the `ProcesNode` instances in the system.

-DEPLOYMENT VIEW-

BP_1 - There is at least one `ProcesNode` in $\mathbb{P}$, e.g. P_x, having a large `QueueLength`. Let us define by F_{maxQL} the function providing the maximum `QueueLength` among all the hardware entities of the processing node. The first condition for the antipattern occurrence is that the value obtained from F_{maxQL} is greater than a threshold Th_{maxQL}:

$$F_{maxQL}(P_x) \geq Th_{maxQL} \tag{3.6}$$

BP_2 - P_x has a heavy computation. This means that the utilizations of some hardware entities in P_x (i.e. cpu, disk) exceed predefined limits. We use the already defined $F_{maxHwUtil}$ to identify the highest `utilization` of cpu(s) and disk(s) in P_x, and then we compare such utilizations to the $Th_{maxCpuUtil}$ and $Th_{maxDiskUtil}$ thresholds:

$$F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \tag{3.7a}$$
$$F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \tag{3.7b}$$

BP_3- The processing nodes are not used in a well-balanced way, as there is at least another instance of `ProcesNode` in $\mathbb{P}$, e.g. P_y, whose `Utilization` of the hardware entities, differentiated according to their type (i.e. cpu, disk), is smaller than the one in P_x. In particular two new thresholds, i.e. $Th_{minCpuUtil}$ and $Th_{minDiskUtil}$, are introduced:

$$F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil} \tag{3.8a}$$
$$F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil} \tag{3.8b}$$

Summarizing, the *Concurrent Processing Systems* antipattern occurs when the following composed predicate is true:

$$\exists P_x, P_y \in \mathbb{P} \mid \boxed{(3.6) \wedge [((3.7a) \wedge (3.8a)) \vee ((3.7b) \wedge ((3.8b)))]}$$

where $\mathbb{P}$ represents the set of all the `ProcesNodes` in the software system. Each (P_x, P_y) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Concurrent Processing Systems antipattern.

Pipe and Filter Architectures and **Extensive Processing** antipatterns are both manifestations of the unbalanced processing: "*Imagine waiting in an airline check-in line. Multiple agents can speed-up the process but, if a customer needs to change an entire itinerary, the agent serving him or her is tied-up for a long time making those changes. With this agent (processor) effectively removed from the pool for the time required to service this request, the entire line moves more slowly and, as more customers arrive, the line becomes longer*" quoted by [121].

PIPE AND FILTER ARCHITECTURES

Problem "Pipe and Filter" Architectures [121] has the following problem informal definition: *"occurs when the slowest filter in a pipe and filter architecture causes the system to have unacceptable throughput"* (see Table 3.1).

"*For example, in the travel analogy, passengers must go through several stages (or filters): first check in at the ticket counter, then pass through security, then go through the boarding process. Recent events have caused each stage to go more slowly. The security stage tends to be the slowest filter these days*" quoted by [121].

We formalize this sentence with four basic predicates: the BP_1, BP_2, BP_3 predicates whose elements belong to the Static View; the BP_4 predicate whose elements belong to the Deployment View.

-STATIC VIEW-

BP_1 - There is at least one `Operation` Op that represents the slowest filter, i.e. it requires a set of resource demands higher than a given thresholds set. Let us define by $F_{resDemand}$ the function providing the `StructuredResourceDemand` of the operation Op that returns an array of values corresponding to the resource demand(s) the operation Op requires:

$$\forall i : F_{resDemand}(Op)[i] \geq Th_{resDemand}[i] \tag{3.9}$$

BP_2 - There is at least one `Service` S that invokes the `OperationInstance` OpI, that is instance of Op. Let us define by $F_{probExec}$ the function that provides the probability of execution of the operation instance OpI when the service S is invoked. If it is equal to 1 it means that the `Task` referring to such operation is mandatory:

$$F_{probExec}(S, OpI) = 1 \tag{3.10}$$

BP_3- The throughput of the service S is unacceptable, i.e. lower than the value defined by an user requirement Th_{SthReq}. Let us define by F_T the function that returns the `Throughput` of the service S:

$$F_T(S) < Th_{SthReq} \tag{3.11}$$

-DEPLOYMENT VIEW-

BP_4- The `ProcesNode` P_{swE_x} on which the software instance swE_x (i.e. the software entity instance that offers OpI) is deployed has a heavy computation. For the formalization of this characteristic, we recall the $F_{maxHwUtil}$ function with the 'all' option, that returns the maximum `Utilization` among all the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \tag{3.12}$$

Summarizing, the *"Pipe and Filter" Architectures* antipattern occurs when the following composed predicate is true:

$$\exists OpI \in \mathbb{O}, S \in \mathbb{S} \mid \boxed{(3.9) \wedge (3.10) \wedge ((3.11) \vee (3.12))}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (OpI, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a "Pipe and Filter" Architectures antipattern.

EXTENSIVE PROCESSING

Problem Extensive processing [121] has the following problem informal definition: *"occurs when extensive processing in general impedes overall response time"* (see Table 3.1).

"*This situation is analogous to the itinerary-change example. It occurs when a long running process monopolizes a processor. The processor is removed from the pool, but unlike the pipe and filter example, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload*" quoted by [121].

We formalize this sentence with four basic predicates: the BP_1, BP_2, BP_3 predicates whose elements belong to the Static View; the BP_4 predicate whose elements belong to the Deployment View.

-STATIC VIEW-

BP_1 - There are at least two `Operations` Op_1 and Op_2 such that: (i) Op_1 has a resource demand vector higher than an upper bound threshold vector (3.13a); (ii) Op_2 has a resource demand vector lower than a lower bound threshold vector (3.13b). The `StructuredResourceDemand` of the operations is provided by the function $F_{resDemand}$ that returns an array of values corresponding to the resource demand(s) the operations Op_1 and Op_2 require. In practice:

$$\forall i : F_{resDemand}(Op_1)[i] \geq Th_{maxOpResDemand}[i] \tag{3.13a}$$

$$\forall i : F_{resDemand}(Op_2)[i] < Th_{minOpResDemand}[i] \tag{3.13b}$$

BP_2 - There is at least one `Service` S that invokes the `OperationInstances` OpI_1 (instance of Op_1), and OpI_2 (instance of Op_2). Unlike to the "Pipe and Filter" Architectures, OpI_1 and OpI_2 are alternately executed in the service S. This condition can be formalized by using the $F_{probExec}$ function that returns the probability of execution of the operation instances OpI_1 and OpI_2 when the service S has been invoked, such that:

$$F_{probExec}(S, OpI_1) + F_{probExec}(S, OpI_2) = 1 \tag{3.14}$$

BP_3- The response time of the service S is unacceptable, i.e. larger than the value Th_{SrtReq} defined by a user requirement. Let us define by F_{RT} the function that returns the `ResponseTime` of the service S:

$$F_{RT}(S) > Th_{SrtReq} \tag{3.15}$$

-DEPLOYMENT VIEW-

BP_4- The `ProcesNode` P_{swE_x} on which the software instance swE_x (i.e. the software entity instance that offers the operation instance OpI_1) is deployed has a heavy computation. For the formalization of this characteristic, we recall the $F_{maxHwUtil}$ function with the 'all' option, that returns the maximum `Utilization` among all the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \tag{3.16}$$

Summarizing, the *Extensive Processing* antipattern occurs when the following composed predicate is true:

$$\exists OpI_1, OpI_2 \in \mathbb{O}, S \in \mathbb{S} \mid \boxed{(3.13a) \wedge (3.13b) \wedge (3.14) \wedge ((3.15) \vee (3.16))}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (OpI_1, OpI_2, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an Extensive Processing antipattern.

CIRCUITOUS TREASURE HUNT

Problem Circuitous Treasure Hunt [123] has the following problem informal definition: *"occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer"* (see Table 3.1).

We formalize this sentence with three basic predicates: the BP_1 predicate whose elements belong to the Static and the Dynamic Views; the BP_2 and BP_3 predicates whose elements belong to the Deployment View.

-STATIC and DYNAMIC VIEWS-

BP_1- There are two `SoftwareEntityInstances`, e.g. swE_x and swE_y, such that: *i)* they are both involved in a `Service` S; *ii)* the instance playing the `senderRole` (e.g. swE_x), sends an excessive number of `Messages` to the one,playing the `receiverRole` (e.g. swE_y); *iii)* the receiver is a database (as captured by the `isDB` attribute). To formalize such interpretation we use the $F_{numDBmsgs}$ function that provides the number of messages sent by swE_x to swE_y in the `Service` S. The property of sending an excessive number of messages can be checked by comparing the output value of the $F_{numDBmsgs}$ function with a threshold $Th_{maxDBmsgs}$:

$$swE_y.isDB = true \tag{3.17}$$

$$F_{numDBmsgs}(swE_x, swE_y, S) \geq Th_{maxDBmsgs} \tag{3.18}$$

-DEPLOYMENT VIEW-

BP_2- The `ProcesNode` P_{swE_y} on which the software instance swE_y is deployed has a heavy computation. That is, the `Utilization` of hardware entities belonging to the `ProcesNode` P_{swE_y} exceed a certain threshold $Th_{maxHwUtil}$. For the formalization of this characteristic, we recall that the $F_{maxHwUtil}$ function with the 'all' option returns the maximum `Utilization` among all the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{swE_y}, all) \geq Th_{maxHwUtil} \tag{3.19}$$

BP_3- Since, in general, a database access utilizes more disk than cpu, we require that the maximum disk(s) `Utilization` is larger than the maximum cpu(s) `utilization` of the `ProcesNode` P_{swE_y}:

$$F_{maxHwUtil}(P_{swE_y}, disk) > F_{maxHwUtil}(P_{swE_y}, cpu) \tag{3.20}$$

Summarizing, the *Circuitous Treasure Hunt* antipattern occurs when the following composed predicate is true:

$$\exists swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(3.17) \land (3.18) \land (3.19) \land (3.20)}$$

where sw$\mathbb{E}$ represents the set of `SoftwareEntityInstances`, and $\mathbb{S}$ represents the set of `Services` in the software system. Each (swE_x, swE_y, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Circuitous Treasure Hunt antipattern.

EMPTY SEMI TRUCKS

Problem Empty Semi Trucks [123] has the following problem informal definition: *"occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both"* (see Table 3.1).

We formalize this sentence with three basic predicates: the BP_1 predicate whose elements belong to the Dynamic View; the BP_2 and BP_3 predicates whose elements belong to the Deployment View.

-DYNAMIC VIEW-

BP_1 - There is at least one `SoftwareEntityInstance` swE_x that exchanges an excessive number of `Messages` with remote software entities. Let us define by $F_{numRemMsgs}$ the function that calculates the number of remote messages sent by swE_x in a `Service` S:

$$F_{numRemMsgs}(swE_x, S) \geq Th_{maxRemMsgs} \tag{3.21}$$

-DEPLOYMENT VIEW-

BP_2- The inefficient use of available bandwidth means that the `SoftwareEntityInstance` swE_x sends a high number of messages without

optimizing the network capacity. Hence, the ProcesNode P_{swE_x}, on which the software entity instance swE_x is deployed, reveals an utilization of the network lower than the threshold $Th_{minNetUtil}$. We focus on the NetworkLink(s) that connect P_{swE_x} to the whole system, i.e. the ones having P_{swE_x} as their EndNode. Since we are interested to the network links on which the software instance swE_x generates traffic, we restrict the whole set of network links to the ones on which the interactions of the software instance swE_x with other communicating entities take place:

$$F_{maxNetUtil}(P_{swE_x}, swE_x) < Th_{minNetUtil} \tag{3.22}$$

BP_3- The inefficient use of interface means that the software instance swE_x communicates with a certain number of remote instances, all deployed on the same remote processing node. Let us define by $F_{numRemInst}$ the function that provides the maximum number of remote instances with which swE_x communicates in the service S. The antipattern can occur when this function returns a value higher or equal than a threshold $Th_{maxRemInst}$:

$$F_{numRemInst}(swE_x, S) \geq Th_{maxRemInst} \tag{3.23}$$

Summarizing, the *Empty Semi Trucks* antipattern occurs when the following composed predicate is true:

$$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(3.21) \wedge ((3.22) \vee (3.23))}$$

where sw$\mathbb{E}$ represents the SoftwareEntityInstances, and $\mathbb{S}$ represents the Services in the software system. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an Empty Semi Trucks antipattern.

TOWER OF BABEL

Problem Tower of Babel [123] has the following problem informal definition: *"occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML"* (see Table 3.1).

We formalize this sentence with two basic predicates: the BP_1 predicate whose elements belong to the Dynamic View; the BP_2 predicate whose elements belong to the Deployment View.

-DYNAMIC VIEW-

BP_1 - There is at least one `Service` S in which the information is often translated from an internal format to an exchange format, and back. Let us define by F_{numExF} the function that counts how many times the `format` is changed in the service S by a `Software-EntityInstance` swE_x. The antipattern can occur when this function returns a value higher or equal than a threshold Th_{maxExF}:

$$F_{numExF}(swE_x, S) \geq Th_{maxExF} \tag{3.24}$$

-DEPLOYMENT VIEW-

BP_2- The `ProcesNode` P_{swE_x} on which the software entity instance swE_x is deployed has a heavy computation. That is, the `Utilization` of hardware entities belonging to the `ProcesNode` P_{swE_x} exceeds a threshold value. For the formalization of this characteristic, we recall the $F_{maxHwUtil}$ function, with the 'all' option, that returns the maximum `Utilization` among the ones of the hardware entities of the processing node:

$$F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \tag{3.25}$$

Summarizing, the *Tower of Babel* antipattern occurs when the following composed predicate is true:

$$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(3.24) \vee (3.25)}$$

where sw$\mathbb{E}$ represents the set of all `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Tower of Babel antipattern.

ONE-LANE BRIDGE

Problem One-Lane Bridge [119] has the following problem informal definition: *"occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g. when accessing a database). Other processes are delayed while they wait for their turn"* (see Table 3.1).

We formalize this sentence with four basic predicates: the BP_1 and BP_2 predicates whose elements belong to the Dynamic View; the BP_3 predicate whose elements belong to the Static View.

-DYNAMIC VIEW-

BP_1 - There is at least one `SoftwareEntityInstance` swE_x that receives a large number of synchronous calls, i.e. its `capacity` (i.e. the parallelism degree) is lower than the incoming requests rate in a service S. Let us define by $F_{numSynchCalls}$ the function providing the number of synchronous calls that swE_x receives for a service S, and by $F_{poolSize}$ the function providing the pool size capacity of swE_x:

$$F_{numSynchCalls}(swE_x, S) \gg F_{poolSize}(swE_x) \tag{3.26}$$

BP_2- The requests incoming to the processing node on which swE_x is deployed are delayed, the `ServiceTime` is much lower than the `WaitingTime`. Let us define by $F_{serviceTime}$ and $F_{waitingTime}$ the functions providing the service time and the waiting time, respectively, for the processing node P_{swE_x}:

$$F_{serviceTime}(P_{swE_x}) \ll F_{waitingTime}(P_{swE_x}) \tag{3.27}$$

-STATIC VIEW-

BP_3- The response time of the service S is unacceptable, i.e. larger than the value Th_{SrtReq} defined by a user requirement. Let us define by F_{RT} the function that returns the `ResponseTime` of the service S:

$$F_{RT}(S) > Th_{SrtReq} \tag{3.28}$$

Summarizing, the ***One-Lane Bridge*** antipattern occurs when the following composed predicate is true:

$$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(3.26) \wedge (3.27) \wedge (3.28)}$$

where sw$\mathbb{E}$ represents the `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a One-Lane Bridge antipattern.

EXCESSIVE DYNAMIC ALLOCATION

Problem Excessive Dynamic Allocation [119] has the following problem informal definition: *"occurs when an application unnecessarily creates and destroys large*

numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance" (see Table 3.1).

We formalize this sentence with two basic predicate: the BP_1 predicate whose elements belong to the Dynamic View, and the BP_2 predicate whose elements belong to the Static View.

-DYNAMIC VIEW-

BP_1 - There is at least one `Service` S in which objects are created in a sort of "just-in-time" approach, when their capabilities are needed, and then destroyed when they are no longer required. Let us define by $F_{numCreatedObj}$ the function that calculates the number of created objects (i.e. `IsCreateObjectAction`), and by $F_{numDestroyedObj}$ the function that returns the number of destroyed ones (i.e. `IsDestroyObjectAction`):

$$F_{numCreatedObj}(S) \geq Th_{maxCrObj} \tag{3.29}$$

$$F_{numDestroyedObj}(S) \geq Th_{maxDeObj} \tag{3.30}$$

-STATIC VIEW-

BP_2 - The overhead for creating and destroying a single object may be small, but when a large number of objects are frequently created and then destroyed, the response time may be significantly increased. Let us recall the function F_{RT} that returns the `ResponseTime` of the service S. If such value is larger than the user requirement, then the antipattern can occur:

$$F_{RT}(S) > Th_{SrtReq} \tag{3.31}$$

Summarizing, the *Excessive Dynamic Allocation* antipattern occurs when the following composed predicate is true:

$$\exists S \in \mathbb{S} \mid \boxed{((3.29) \lor (3.30)) \land (3.31)}$$

where $\mathbb{S}$ represents the set of all the `Services` in the software system. Each S instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an Excessive Dynamic Allocation antipattern.

3.4.2 MULTIPLE-VALUES PERFORMANCE ANTIPATTERNS

In this Section we report the *Performance Antipatterns* that to be detected require the trend (or evolution) of the performance indices along the time (i.e. multiple-values). The numerical values of the performance indices (e.g. the operation response time) come from the simulation of the performance model.

TRAFFIC JAM

Problem Traffic Jam [120] has the following problem informal definition: *"occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared"* (see Table 3.1).

We formalize this sentence with three basic predicates: the BP_1, BP_2, and BP_3 predicates whose elements belong to the Static View.

-STATIC VIEW-

BP_1- There is at least one `OperationInstance` OpI that has a quite stable value of its response time along different observation time interval up to the *k*-th one. Let us define by F_{RT} the function that returns the mean `ResponseTime` of the `OperationInstance` OpI observed in the interval t. We consider the average response time increase of the operation in $k-1$ consecutive time slots in which no peaks are shown, that means it is lower than a threshold $Th_{OpRtVar}$:

$$\frac{\sum_{1 \le t \le k} |(F_{RT}(OpI, t) - F_{RT}(OpI, t-1))|}{k-1} < Th_{OpRtVar} \tag{3.32}$$

BP_2- The `OperationInstance` OpI has an increasing value of its response time along the *k*-th observation interval. We consider the average response time increase of the operation in the k-th time slot in which a peak is shown, that means it is higher than a threshold $Th_{OpRtVar}$:

$$|F_{RT}(OpI, k) - F_{RT}(OpI, k-1)| > Th_{OpRtVar} \tag{3.33}$$

BP_3- The `OperationInstance` OpI has a quite stable value of its response time after the *k*-th observation interval, since the wide variability persists long. Different observation time slots are considered up to the *n*-th observation interval. We consider the average response time increase of the operation in $n-k$ consecutive time slots in which no peaks are shown:

$$\frac{\sum_{k \le t \le n} |(F_{RT}(OpI, t) - F_{RT}(OpI, t-1))|}{n-k} < Th_{OpRtVar} \tag{3.34}$$

Summarizing, the *Traffic Jam* antipattern occurs when the following composed predicate is true:

$$\exists OpI \in \mathbb{O} \mid \boxed{(3.32) \wedge (3.33) \wedge (3.34)}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances` in the software system. Each OpI instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Traffic Jam antipattern.

THE RAMP

Problem The Ramp [123] has the following problem informal definition: *"occurs when processing time increases as the system is used"* (see Table 3.1).

We formalize this sentence with two basic predicates: the BP_1 and BP_2 predicates whose elements belong to the Static View.

-STATIC VIEW-

BP_1- There is at least one `OperationInstance` OpI that has an increasing value for the response time along different observation time slots. Let us define by F_{RT} the function that returns the mean `ResponseTime` of the operation instance OpI observed in the time slot t. The Ramp can occur when the average response time of the operation instance increases in n consecutive time slots, that means it is higher than a threshold $Th_{OpRtVar}$:

$$\frac{\sum_{1 \leq t \leq n} |F_{RT}(OpI, t) - F_{RT}(OpI, t-1)|}{n} > Th_{OpRtVar} \tag{3.35}$$

BP_2- The `OperationInstance` OpI shows a decreasing value for the throughput along different observation time slots. Let us define by F_T the function that returns the mean `Throughput` of the operation instance OpI observed in the time slot t. The Ramp occurs when the absolute value of the average throughput of the operation instance increases in n consecutive time slots, that means it is higher than a threshold $Th_{OpThVar}$:

$$\frac{\sum_{1 \leq t \leq n} |F_T(OpI, t) - F_T(OpI, t-1)|}{n} > Th_{OpThVar} \tag{3.36}$$

Summarizing, *The Ramp* antipattern occurs when the following composed predicate is true:

$$\exists OpI \in \mathbb{O} \mid \boxed{(3.35) \wedge (3.36)}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances` in the software system. Each OpI instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents The Ramp antipattern.

MORE IS LESS

Problem More Is Less [121] has the following problem informal definition: *"occurs when a system spends more time "thrashing" than accomplishing real work because there are too many processes relative to available resources"* (see Table 3.1).

We formalize this sentence with one basic predicate: the BP_1 predicate whose elements belong to the Deployment View.

-DEPLOYMENT VIEW-

BP_1 - There is at least one `ProcesNode` P_x whose configuration parameters are not able to support the workload required to the software system. The parameters we refer are: the number of concurrent *dbConnections*, the *webConnections*, the *pooledResources*, or the *concurrentStreams*.

Let us define by $F_{par}[i]$ the function that returns the i-th configuration parameter defined for the system; and by $F_{RTpar}[i]$ the function returning the i-th run time parameter observed in the time slot t. The More Is Less antipattern can occur when the configuration parameters are much lower than the average values of the run time parameters in n consecutive time slots:

$$\forall i : F_{par}(P_x)[i] \ll \frac{\sum_{1 \leq t \leq n}(F_{RTpar}(P_x, t)[i] - F_{RTpar}(P_x, t-1)[i])}{n} \tag{3.37}$$

Summarizing, the *More Is Less* antipattern occurs when the following predicate is true:

$$\exists P_x \in \mathbb{P} \mid \boxed{(3.37)}$$

where $\mathbb{P}$ represents the set of all the `ProcesNodes` in the software system. Each P_x instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a More Is Less antipattern.

3.4.3 Summary

This Section summarizes the *functions* and the *thresholds* introduced for representing antipatterns as logical predicates.

Table 3.2 reports the supporting functions we need to evaluate software model elements. In particular, the first column of the Table shows the *signature* of the function and the second column provides its *description*. For example, in the first row we can find the $F_{numClientConnects}$ function: it takes as input one software entity instance and it returns an integer that represents the multiplicity of the relationships where swE_x assumes the client role.

Table 3.3 reports the thresholds we need to evaluate software boundaries. In particular, the first column of the Table shows the name of the *threshold*, the second column provides its *description*, and finally it is proposed an *estimation* for binding its numerical value. For example, in the first row we can find the $Th_{maxConnects}$ threshold: it represents the maximum bound for the number of usage relationships a software entity is involved in; it can be estimated as the average number of usage relationships, with reference to the entire set of software instances in the software system, plus the corresponding variance.

Similarly to Table 3.3, Table 3.4 reports the thresholds we need to evaluate hardware boundaries. For example, in the first row we can find the $Th_{maxHwUtil}$ threshold: it represents the maximum bound for the hardware device utilization and it can be estimated as the average number of all the hardware utilization values, with reference to the entire set of hardware devices in the software system, plus the corresponding variance.

Table 3.5 reports the thresholds we need to evaluate service boundaries. In particular, the first column of the Table shows the name of the *threshold* and the second column provides its *description*. For example, in the first row we can find the Th_{SthReq} threshold that represents the required value for the throughput of the service S. These types of thresholds do not require an estimation process since we expect that they are defined by software designers in the requirements specification phase.

Similarly to Table 3.5, Table 3.6 reports the thresholds we need to evaluate operation slopes along the time. For example, in the first row we can find the $Th_{OpRtVar}$ threshold that represents the maximum feasible slope of the response time observed in n consecutive time slots for the operation Op. These type of thresholds do not require an estimation process since we expect that they are defined by software designers.

Finally, Table 3.7 lists the logic-based representation of the performance antipatterns we propose. Each row represents a specific antipattern that is characterized by two attributes: *antipattern* name, and its *formula*, i.e. the first order logics predicate modeling the corresponding antipattern problem.

The formalization of antipatterns is the result of multiple formulations and checks. This is a first attempt to formally define antipatterns and it may be subject to some refine-

Signature	Description
int $F_{numClientConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships where swE_x assumes the client role
int $F_{numSupplierConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships where swE_x assumes the supplier role
int $F_{numMsgs}$ (SoftwareEntityInstance swE_x, SoftwareEntityInstance swE_y, Service S)	It counts the number of messages sent from swE_x to swE_y in a service S
float $F_{maxHwUtil}$ (ProcesNode pn_x, type T)	It provides the maximum hardware utilization among the hardware devices of a certain type T={cpu, disk, all} composing the processing node pn_x
float $F_{maxNetUtil}$ (ProcesNode pn_x, ProcesNode pn_y)	It provides the maximum utilization among the network links joining the processing nodes pn_x and pn_y
float $F_{maxNetUtil}$ (ProcesNode pn_x, SoftwareEntityInstance swE_x)	It provides the maximum utilization among the network links connecting pn_x overall the processing nodes with which swE_x generates traffic
float F_{maxQL} (ProcesNode pn_x)	It provides the maximum queue length among the hardware devices composing the processing node pn_x
int[ ] $F_{resDemand}$ (Operation Op)	It provides the resource demand of the operation Op
float $F_{probExec}$ (Service S, Operation Op)	It provides the probability the operation Op is executed in the service S
float F_T (Service S)	It provides the estimated throughput of the service S at the steady-state
float F_{RT} (Service S)	It provides the estimated response time of the service S at the steady-state
float F_T (Service S, timeInterval t)	It provides the estimated throughput of the service S at the time interval t
float F_{RT} (Service S, timeInterval t)	It provides the estimated response time of the service S at the time interval t
int $F_{numDBmsgs}$ (SoftwareEntityInstance swE_x, SoftwareEntityInstance swE_y, Service S)	It counts the number of requests by swE_x towards swE_y (i.e. a database instance) in a service S
int $F_{numRemMsgs}$ (SoftwareEntityInstance swE_x, Service S)	It counts the number of remote messages sent by swE_x in a service S
int $F_{numRemInst}$ (SoftwareEntityInstance swE_x, Service S)	It provides the number of remote instances with which swE_x communicates in a service S
int F_{numExF} (SoftwareEntityInstance swE_x, Service S)	It provides the number of exchange formats performed by swE_x in a service S
int $F_{numSynchCalls}$ (SoftwareEntityInstance swE_x, Service S)	It provides the number of synchronous calls swE_x receives in a service S
int $F_{poolSize}$ (SoftwareEntityInstance swE_x)	It provides the pool size capacity of swE_x
float $F_{serviceTime}$ (ProcesNode pn_x)	It provides the service time of pn_x
float $F_{waitingTime}$ (ProcesNode pn_x)	It provides the waiting time of pn_x
int $F_{numCreatedObj}$ (Service S)	It provides the number of created objects in a service S
int $F_{numDestroyedObj}$ (Service S)	It provides the number of destroyed objects in a service S
int[ ] F_{par} (ProcesNode pn_x)	It provides the array of configuration parameters related to the pn_x processing node
int[ ] F_{RTpar} (ProcesNode pn_x, timeInterval t)	It provides the array of configuration parameters related to the pn_x processing node at the time interval t

Table 3.2: Functions specification.

Threshold	Description	Estimation
$Th_{maxConnects}$	It represents the maximum bound for the number of usage relationships a software entity is involved	It can be estimated as the average number of usage relationships, with reference to the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxMsgs}$	It represents the maximum bound for the number of messages sent by a software entity in a service	It can be estimated as the average number of sent messages, with reference to the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxDBmsgs}$	It represents the maximum bound for the number of database requests in a service	It can be estimated as the average number of database requests, with reference to the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxRemMsgs}$	It represents the maximum bound for the number of remote messages in a service	It can be estimated as the average number of remote messages, with reference to the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxRemInst}$	It represents the maximum bound for the number of remote communicating instances in a service	It can be estimated as the average number of remote communicating instances, with reference to the entire set of software instances in the software system, plus the corresponding variance
Th_{maxExF}	It represents the maximum bound for the number of exchange formats	It can be estimated as the average number of exchanging formats, with reference to the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxResDemand}[i]$	It represents the maximum bound for the resource demand of operations	It can be estimated as the average number of resource demands, with reference to the entire set of software operations in the software system, plus the corresponding variance
$Th_{minResDemand}[i]$	It represents the minimum bound for the resource demand of operations	It can be estimated as the $Th_{maxOpResDemand}$ minus the gap decided by software architects for resource demand of operations
$Th_{maxCrObj}$	It represents the maximum bound for the number of created objects	It can be estimated as the average number of created objects, with reference to the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxDeObj}$	It represents the maximum bound for the number of destroyed objects	It can be estimated as the average number of destroyed objects, with reference to the entire set of software instances in the software system, plus the corresponding variance

Table 3.3: Thresholds specification: software characteristics.

Threshold	Description	Estimation
$Th_{maxHwUtil}$	It represents the maximum bound for the hardware device utilization	It can be estimated as the average number of all the hardware utilization values, with reference to the entire set of hardware devices in the software system, plus the corresponding variance
$Th_{maxNetUtil}$	It represents the maximum bound for the network link utilization	It can be estimated as the average number of all the used bandwidth values, with reference to the entire set of network links in the software system, plus the corresponding variance
$Th_{minNetUtil}$	It represents the minimum bound for the network link utilization	It can be estimated as the $Th_{maxNetUtil}$ minus the gap decided by software architects for network links
Th_{maxQL}	It represents the maximum bound for the queue length utilization	It can be estimated as the average number of all the queue length values, with reference to the entire set of hardware devices in the software system, plus the corresponding variance
$Th_{maxCpuUtil}$	It represents the maximum bound for the cpu utilization	It can be estimated as the mean value of all the hardware utilization values, with reference to the entire set of cpu resources in the software system, plus the corresponding variance
$Th_{maxDiskUtil}$	It represents the maximum bound for the disk utilization	It can be estimated as the mean value of all the hardware utilization values, with reference to the entire set of disk resources in the software system, plus the corresponding variance
$Th_{minCpuUtil}$	It represents the minimum bound for the cpu utilization	It can be estimated as the $Th_{maxCpuUtil}$ minus the gap decided by software architects for cpu devices
$Th_{minDiskUtil}$	It represents the minimum bound for the disk utilization	It can be estimated as the $Th_{maxDiskUtil}$ minus the gap decided by software architects for disk devices

Table 3.4: Thresholds specification: hardware characteristics.

Threshold	Description
Th_{SthReq}	It represents the required value for the throughput of the service S
Th_{SrtReq}	It represents the required value for the response time of the service S

Table 3.5: Thresholds specification: requirements.

Threshold	Description
$Th_{OpRtVar}$	It represents the maximum feasible slope of the response time observed in n consecutive time slots for the operation Op
$Th_{OpThVar}$	It represents the maximum feasible slope of the throughput observed in n consecutive time slots for the operation Op

Table 3.6: Thresholds specification: slopes.

ments. However, the logic-based formalization is meant to demonstrate the potential for a machine-processable management of performance antipatterns.

In fact, as proof of concept, a first implementation of the Detection Engine (see Figure 3.1) has been made with a Java application automating the check of the architectural model properties, as stated in the logical predicates. Such application parses any XML document compliant with our XML Schema and returns the instances of the detected antipatterns[4].

3.5 Towards the specification of the antipattern solution

With the logic-based formalization we have worked towards the detection of antipatterns. Since an antipattern is made of a problem description as well as a solution description, in this Section we work on the solution representation.

The basic idea is to exploit the formalization provided in Section 3.4 to automatically deduce antipattern solution. Just to give a hint, being an antipattern expressed as a logical formula, the negation of such formula should provide some suggestions on the refactoring actions to solve the antipattern. This could represent a general approach to automatically solve any new type of antipattern that can be defined in future, or at least to provide guidelines that better support the antipattern solution.

In the following we report the formalization of the *solution* for the Blob antipattern by negating its logical formula.

[4]The detection engine uses the Java API for XML processing, that is Document Object Model (DOM) [9], i.e. a cross-platform and language-independent convention for representing and interacting with objects in XML documents.

	Antipattern		Formula
Single-value	Blob (or god class/component)		$\exists swE_x, swE_y \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid (F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \vee F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects}) \wedge (F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \vee F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs}) \wedge (F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \vee F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil})$
	Unbalanced Processing	Concurrent Processing Systems	$\exists P_x, P_y \in \mathbb{P} \mid F_{maxQL}(P_x) \geq Th_{maxQL} \wedge [(F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \wedge F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil}) \vee (F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \wedge (F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil}))]$
		“Pipe and Filter” Architectures	$\exists OpI \in \mathbb{O}, S \in \mathbb{S} \mid \forall i : F_{resDemand}(Op)[i] \geq Th_{resDemand}[i] \wedge F_{probExec}(S, OpI) = 1 \wedge (F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \vee F_T(S) < Th_{SthReq})$
		Extensive Processing	$\exists OpI_1, OpI_2 \in \mathbb{O}, S \in \mathbb{S} \mid \forall i : F_{resDemand}(Op_1)[i] \geq Th_{maxOpResDemand}[i] \wedge \forall i : F_{resDemand}(Op_2)[i] < Th_{minOpResDemand}[i] \wedge F_{probExec}(S, OpI_1) + F_{probExec}(S, OpI_2) = 1 \wedge (F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \vee F_{RT}(S) > Th_{SrtReq})$
	Circuitous Treasure Hunt		$\exists swE_x, swE_y \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid swE_y.isDB = true \wedge F_{numDBmsgs}(swE_x, swE_y, S) \geq Th_{maxDBmsgs} \wedge F_{maxHwUtil}(P_{swE_y}, all) \geq Th_{maxHwUtil} \wedge F_{maxHwUtil}(P_{swE_y}, disk) > F_{maxHwUtil}(P_{swE_y}, cpu)$
	Empty Semi Trucks		$\exists swE_x \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid F_{numRemMsgs}(swE_x, S) \geq Th_{maxRemMsgs} \wedge F_{maxNetUtil}(P_{swE_x}, swE_x) < Th_{minNetUtil} \vee F_{numRemInst}(swE_x, S) \geq Th_{maxRemInst})$
	Tower of Babel		$\exists swE_x \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid F_{numExF}(swE_x, S) \geq Th_{maxExF} \vee F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil}$
	One-Lane Bridge		$\exists swE_x \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid F_{numSynchCalls}(swE_x, S) \gg F_{poolSize}(swE_x) \wedge F_{serviceTime}(P_{swE_x}) \ll F_{waitingTime}(P_{swE_x}) \wedge F_{RT}(S) > Th_{SrtReq}$
	Excessive Dynamic Allocation		$\exists S \in \mathbb{S} \mid (F_{numCreatedObj}(S) \geq Th_{maxCrObj} \vee F_{numDestroyedObj}(S) \geq Th_{maxDeObj}) \wedge F_{RT}(S) > Th_{SrtReq}$
Multiple-values	Traffic Jam		$\exists OpI \in \mathbb{O} \mid \frac{\sum_{1 \leq t \leq k} \lvert (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \rvert}{k-1} < Th_{OpRtVar} \wedge F_{RT}(OpI,k) - F_{RT}(OpI,k-1) > Th_{OpRtVar} \wedge \frac{\sum_{k \leq t \leq n} \lvert (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \rvert}{n-k} < Th_{OpRtVar}$
	The Ramp		$\exists Op \in \mathbb{O} \mid \frac{\sum_{1 \leq t \leq n} \lvert (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \rvert}{n} > Th_{OpRtVar} \wedge \frac{\sum_{1 \leq t \leq n} \lvert (F_{T}(OpI,t) - F_{T}(OpI,t-1)) \rvert}{n} > Th_{OpThVar}$
	More is Less		$\exists P_x \in \mathbb{P} \mid \forall i : F_{par}(P_x)[i] \ll \frac{\sum_{1 \leq t \leq N} (F_{RTpar}(P_x,t)[i] - F_{RTpar}(P_x,t-1)[i])}{N}$

Table 3.7: A logic-based representation of Performance Antipatterns.

BLOB (OR GOD CLASS/COMPONENT)

Solution Blob (or "god" class/component) [119] has the following solution informal definition: *"Refactor the design to distribute intelligence uniformly over the applications top-level classes, and to keep related data and behavior together"* (see Table 3.1).

Let us try to model the solution of the antipattern by negating the predicates formalizing the antipattern *problem* definition. Hence, the *Blob (or "god" class/-component)* antipattern should be solved when: $\forall swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid$ $\boxed{\neg(((3.1) \vee (3.2)) \wedge ((3.3a) \vee (3.3b)) \wedge ((3.4) \vee (3.5)))}$.

By applying De Morgan's law (i.e. the negation of a conjunction is the disjunction of the negations) the previous formula can be written as follows: $\forall swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid$ $\boxed{\neg((3.1) \vee (3.2)) \vee \neg((3.3a) \vee (3.3b)) \vee \neg((3.4) \vee (3.5))}$.

By applying De Morgan's law (i.e. the negation of a disjunction is the conjunction of the negations) the previous formula can be written as follows: $\forall swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid$ $\boxed{\neg(3.1) \wedge \neg(3.2) \vee \neg(3.3a) \wedge \neg(3.3b) \vee \neg(3.4) \wedge \neg(3.5)}$.

Figure 3.16 focuses on the static view of the *Blob-controller* antipattern, and some reconfiguration actions that can be applied to solve such antipattern are depicted. The upper side of Figure 3.16 refers to the first approximation we gave in Section 3.3 (see Figure 3.2), whereas the lower side shows that other options discussed in the following.

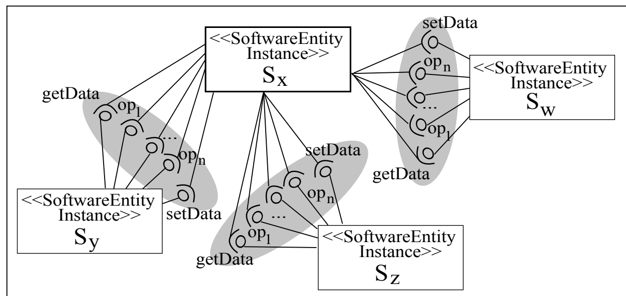
$\neg$(3.1) $\wedge$ $\neg$(3.2) - There is no `SoftwareEntityInstance` such that it performs most of the work of the system and holds all the application's data, relegating other classes or components to minor and supporting roles.

Starting from the assumption that swE_x has been detected as a Blob instance and it has a *high* number of connections, a solution might be to delegate some work from swE_x to its surrounding software entities, thus to avoid an excessive communication. Let us recall the threshold $Th_{maxConnects}$ as the upper bound for the number of the connections. It means that the software instance swE_x has a number of n connections, and obviously $n > Th_{maxConnects}$. The problem becomes how to move $\lceil n - (Th_{maxConnects} - 1)\rceil$ connections from swE_x to its surrounding instances so that each software instance has a maximum number of $(Th_{maxConnects} - 1)$ connections.

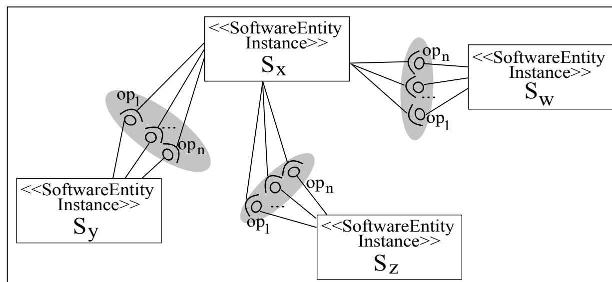
A first strategy (see *Reconfiguration (a)* of Figure 3.16) could be to split the software instance in about $\lceil n/(Th_{maxConnects} - 1)\rceil$ software instances, i.e. swE_{x1}, $\ldots$, $swE_{x\lceil n/(Th_{maxConnects}-1)\rceil}$, and for each new software instance maintaining $(Th_{maxConnects} - 1)$ connections.

A second strategy (see *Reconfiguration (b)* of Figure 3.16) could be to better separate the concerns by reasoning on the number of the software instances with which swE_x is connected. Let us define $SUBs_(swE_x) = \{swE_{y1}, \ldots, swE_{yk}\}$. The number of

"BLOB-controller" problem



"BLOB-controller" solution



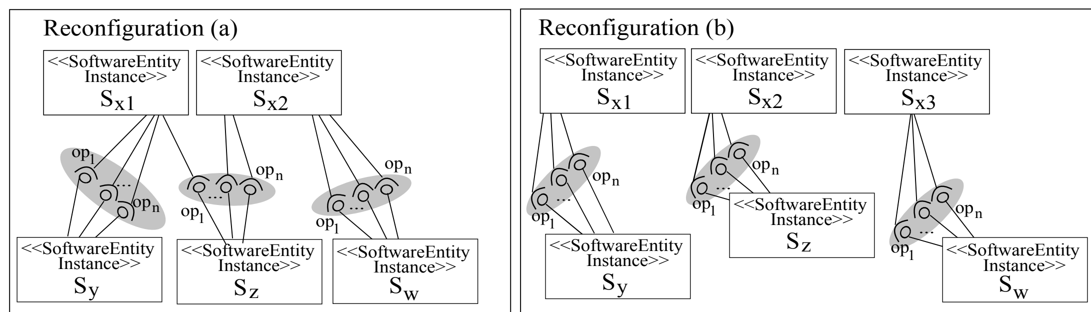


Figure 3.16: A graphical representation of the reconfiguration actions for solving the *Blob-controller* Antipattern.

connections between swE_x and swE_{yi}, with $i = \{1, \ldots, k\}$ is $c_1, \ldots, c_k$. For each swE_{yi}, if $c_k < Th_{maxConnects}$ the software instance swE_x is split in swE_{xi} and connected with the swE_{yi}, otherwise it is necessary to split the software instance in $\lceil c_k/(Th_{maxConnects} - 1) \rceil$ software instances, i.e. $swE_{x1}, \ldots, swE_{x\lceil c_k/(Th_{maxConnects}-1)\rceil}$, and for each new software instance maintaining $(Th_{maxConnects} - 1)$ connections. The final goal is that the software instance swE_x has not anymore a number of connections larger than the threshold value.

¬(3.3a) ∧ ¬(3.3b) - There is no `SoftwareEntityInstance` such that it generates and receives excessive message traffic.

Starting from the assumption that swE_x has been detected as a Blob instance and it sends or receives a *high* number of messages, a solution might be to reduce the number of messages, thus to avoid an excessive traffic. Let us recall the threshold $Th_{maxMsgs}$ as the upper bound for the number of the messages. It means that the software instance swE_x sends or receives a number of n messages, and obviously $n > Th_{maxMsgs}$. It is necessary to delete $\lceil n - (Th_{msgs} - 1) \rceil$ messages by looking at the behavior and checking how to move the business logics among the software instances.

¬(3.4) ∧ ¬(3.5) - There is no a `ProcesNode` and a `NetworkLink` such that they have a *high* utilization.

Decreasing the number of connections and messages might help to improve the utilization of both processing nodes and network links. However, if the software instance swE_x has been split in n instances, $swE_{x1}, \ldots, swE_{xn}$, an additional solution might be to re-deploy such instances on the available hardware resources, thus to balance the workload. Let us recall the threshold $Th_{maxHwUtil}$ as the upper bound for the utilization of processing nodes, and $Th_{maxNetUtil}$ as the upper bound for the utilization of network links. A strategy could be to define as $Available_PNs = \{P_1, \ldots, P_m\} \mid \forall i : F_{maxHwUtil}(P_i, all) < Th_{maxHwUtil}$, and $Available_NLs = \{N_1, \ldots, N_k\} \mid \forall i, j \in Available_PNs : F_{maxNetUtil}(P_i, P_j) < Th_{maxNetUtil} \wedge \forall x \in\mid Available_NLs : N_x.endNode = \{P_i, P_j\}$. The resource demand vector of each software entity instance and their interactions give the information about how to re-deploy the instances. High values of resource demands can be assigned to processing nodes with low utilization, and instances with many interactions can be assigned to the same processing node thus to avoid remote communications.

The negation of logical predicates represents a first attempt towards the automation of the antipatterns solution, in fact it only provides guidelines for refactoring sofware architectural models. As described above, thresholds are fundamental to define the refactoring actions, and several strategies can be devised by reasoning on such values.

However, we believe that a wider investigation is necessary to formally define the refactoring actions for solving the antipatterns. In fact the approach of negating logical predicates will not be used in Chapter 5 where the solution of antipatterns in concrete modeling languages has been discussed by reasoning on their textual informal specification.

CHAPTER 4

SPECIFYING ANTIPATTERNS: A MODEL-DRIVEN APPROACH

This Chapter mainly advocates the idea of modeling antipatterns as first-class entities, i.e. as models that can be manipulated to conveniently generate further artifacts with model-driven techniques. A Performance Antipattern Modeling Language (PAML) is provided for specifying antipatterns, we show how to model antipatterns with it, and the benefits of the approach are finally discussed. In other words, the PAML models of antipatterns characterize the elements and the semantic properties we must search in the software architectural models to detect antipattern occurrences.

The advantage of introducing a modeling language for specifying antipatterns is to let the designer precisely convey the intended interpretation on the basis of their informal definitions and possibly its subsequent emendations. The core question tackled in this Chapter is: how can performance antipatterns be unambiguously specified by a designer? To this aim, a model-driven approach is introduced to allow a machine-processable and user-friendly specification of performance antipatterns.

4.1 PERFORMANCE ANTIPATTERNS MODELING LANGUAGE (PAML)

Model Driven Engineering [115] (MDE) leverages intellectual property and business logic from source code into high-level specifications enabling more accurate analyses. In general, an application domain is consistently analyzed and engineered by means of a metamodel, i.e. a coherent set of interrelated concepts.

This Section presents a metamodel, named Performance Antipattern Modeling Language (PAML), collecting all the architectural model elements identified by analyzing the antipatterns definition in literature [123], and fully described in Appendix A.

The PAML metamodel structure is shown in Figure 4.1. It is constituted by two main parts as delimited by the horizontal dashed line: (i) the *Antipattern Specification* part is aimed at collecting the high-level features, such as the views of the system (i.e. static, dynamic,

deployment), characterizing the antipattern specification; (ii) the *Model Elements Specification* part is aimed at collecting the concepts used to represent the software architectural models and the performance results, that are the ones on which the antipatterns are based.

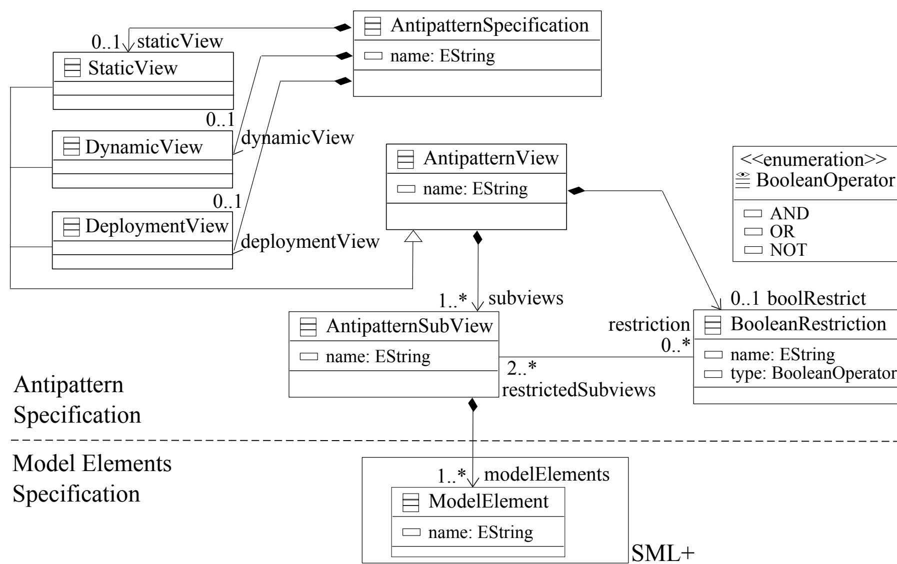


Figure 4.1: The Performance Antipattern Modeling Language (PAML) structure.

Note that PAML currently only deals with the definition of performance problems captured by antipatterns. As future work we plan to complete PAML with a Refactoring Modeling Language (RML) for formalizing the solutions in terms of refactorings, i.e. changes of the original software architectural model.

All the architectural model elements and the performance indices occurring in the antipattern specifications are grouped in a sub-metamodel called SML+ (see Figure 4.2). SML+ obviously shares many concepts with existing Software Modeling Languages, however, it is not meant to be another modeling language, rather it is oriented to specify the basic elements of performance antipatterns.

We do not detail all model elements of SML+ since they reflect the concepts of the XML Schema presented in Appendix A. Dashed boxes of Figure 4.2 identify the three packages representing software architectural model features, i.e. the Static View, Dynamic View, Deployment View. The *Static View Model Elements* package contains the software elements (e.g. `SoftwareEntity`, `Operation`, etc.) involved in the system and the static `Relationships` among them (e.g. *client*, *supplier*) that are aimed at defining the system software resources. The *Dynamic View Model Elements* package contains the interaction elements (e.g. `Behavior`, `Message`, etc.) involved in the system and the dynamic relationships among them (e.g. *behaviors*, *msgs*, etc.) that are aimed at providing the system functionalities. Finally, the *Deployment View Model Elements* package contains the platform elements (e.g. `ProcesNode`, `NetworkLink`, etc.) involved in the system and the deployment relationships (e.g. *endNode*, *hwResTypes*, etc.) that are aimed at defining the system platform resources and the deployment policies.

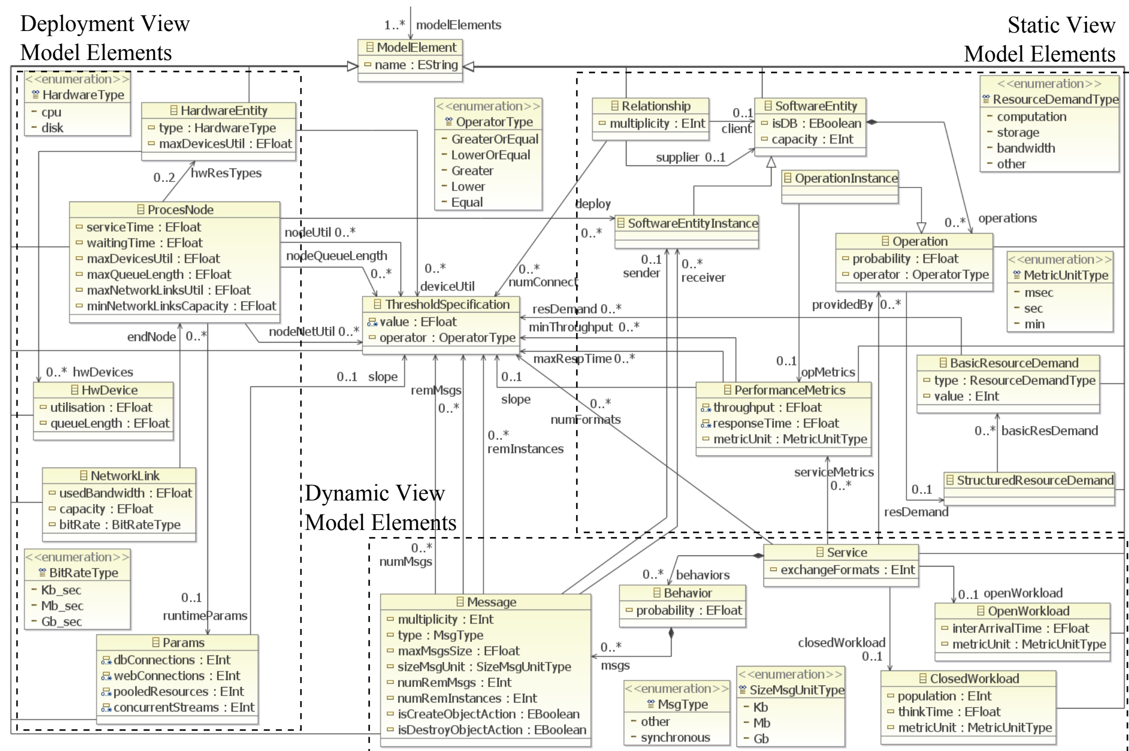


Figure 4.2: The Enriched Software Modeling Language (SML+).

Several relationships are also defined among different views, e.g. the `SoftwareEntityInstance` meta-class is involved in a *deploy* relationship with the `ProcesNode` meta-class to denote the deployment of software instances into hardware platforms.

Out of the three packages, in Figure 4.2 the *ThresholdSpecification* meta-class is defined. It has an identifier *name*, one or more float *value*s representing the numerical bindings, and finally an *operator* drives the designer towards the interpretation of the threshold through the *OperatorType* that may be: GreaterOrEqual ($\geq$), LowerOrEqual ($\leq$), Greater ($>$), Lower ($<$), Equal ($=$).

PAML within the Eclipse toolkit

PAML is implemented within the Eclipse Modeling Framework (EMF) [3], i.e. a modeling framework and code generation facility for building tools and other applications based on a structured data model. Figure 4.3 shows PAML as expressed in Ecore[1] (*paml.ecore*), i.e. the meta-metamodel included in the core of the EMF framework. EMF is also supported by a validation framework that provides capabilities used to ensure model integrity and, as shown in Figure 4.3, the validation of the PAML metamodel has been successfully completed.

Antipatterns modeling is supported by an Eclipse editor. By default, it is split between two plug-ins: (i) an *edit* plug-in (*projPAML.edit*, see Figure 4.3) that includes adapters

[1] For more details please refer to the Package org.eclipse.emf.ecore [8].

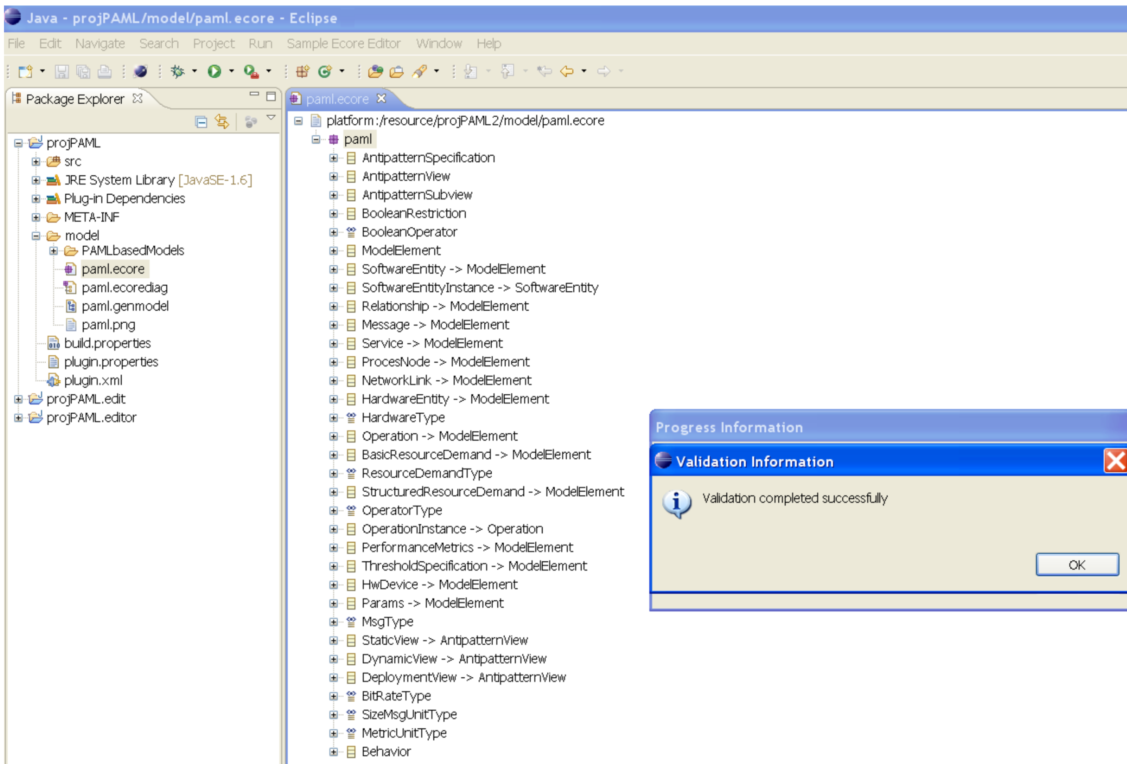


Figure 4.3: PAML implementation in the Eclipse platform.

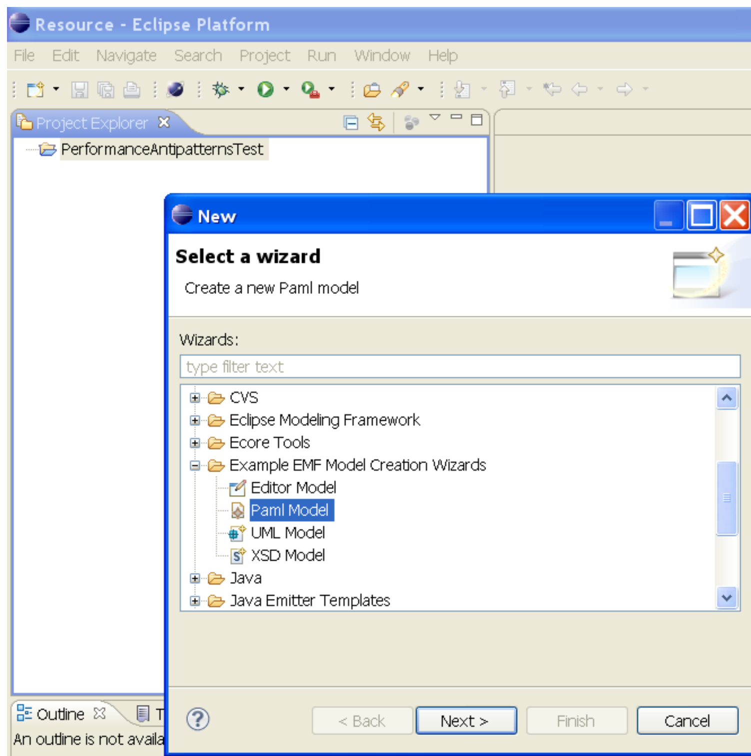


Figure 4.4: PAML Model wizard for creating new instances.

providing a structured view and perform command-based editing of the model objects; (ii) an *editor* plug-in (*projPAML.editor*, see Figure 4.3) that provides the user interface for the editor and the wizard. In order to test the new plug-ins, a second instance of Eclipse must be launched. The plug-ins (i.e. the PAML Model, the PAML Edit, and the PAML Editor) will run in this workbench. The PAML Model wizard can now be used to create a new instance of the model, as shown in Figure 4.4.

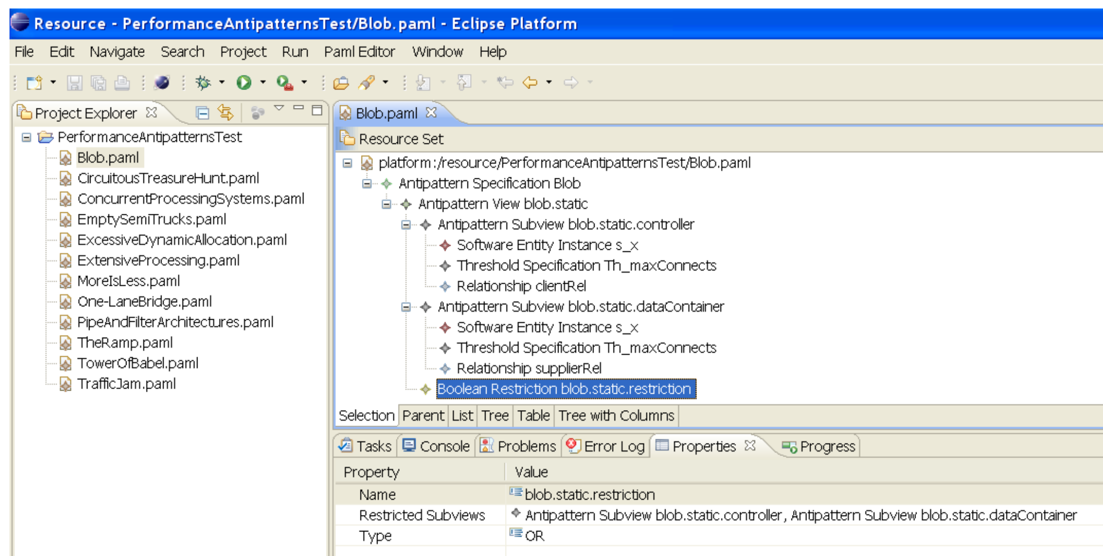


Figure 4.5: PAML Editor properties.

PAML models will be named with the antipattern names, and they will end with a *.paml* extension. Figure 4.5 shows the set of all the antipatterns we modeled, e.g. the Blob (*Blob.paml*), the Circuitous Treasure Hunt (*CircuitousTreasureHunt.paml*), etc.

For example, following the graphical representation of the Blob antipattern (see Figures 3.2 and 3.3), the corresponding Blob model (see Figure 4.5) will be constituted by an `AntipatternSpecification` with three `AntipatternViews`: the `StaticView`; the `DynamicView` and the `DeploymentView`. In particular, in Figure 4.5, an excerpt of the antipattern specification *Blob* is depicted. The Static View (*blob.static*) includes two `AntipatternSubViews` specifying: (i) the BLOB-controller case (*blob.static.controller*); (ii) the BLOB-dataContainer case (*blob.static.dataContainer*). Each subview will contain a set of `ModelElements`, e.g. `SoftwareEntityInstance`, `ThresholdSpecification`, `Relationship`. A `BooleanRestriction` (*blob.static.restriction*) is defined between these sub-views, and the *type* is set by the `BooleanOperator` equal to the *OR* value (see Figure 4.5). We can conclude that the PAML editor represents an instrument to formally specify antipatterns and it is given to the designer to precisely convey his/her interpretation.

4.2 A MODEL-BASED SPECIFICATION OF THE ANTIPATTERN PROBLEM

In this Section we provide the PAML-based models that reflect the interpretation we gave in Chapter 3. PAML-based models are structured (in the Figures of this Section) in two parts: in the upper part we report a fragment of the software architectural model with the occurrence of the antipattern problem (as shown in Section 3.3); in the lower part the PAML-based antipattern model is depicted, and the binding towards the software architectural model elements is highlighted. For sake of readability, in the Figures we only report some bindings (i.e. dashed arrows labeled *BINDS TO*), and antipattern models are fully explained in their textual description.

4.2.1 SINGLE-VALUE PERFORMANCE ANTIPATTERNS

In this Section we report the *Performance Antipatterns* that can be detected by single values of performance indices (such as mean, max or min values).

BLOB (OR GOD CLASS)

Figure 4.6 shows the PAML-based model for the Blob (or god class) antipattern. It contains three views: `StaticView`, `DynamicView` and `DeploymentView`.

The static view is named *blob.static* and it contains two `SubViews`. The first sub-view is named *blob.static.controller*, it captures the BLOB-controller case in which a `SoftwareEntityInstance` S_x is involved in a `Relationship` as *client*, and the *multiplicity* is calculated with the function $F_{numClientConnects}$ (see Table 3.2). The *numConnect* attribute refers to the `ThresholdSpecification` named $Th_{maxConnects}$ (see Table 3.3), and it means that S_x might be a blob antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value. The second sub-view is *blob.static.dataContainer*, it captures the BLOB-dataContainer case in which a `SoftwareEntityInstance` S_x is involved in a `Relationship` as *supplier*, and the *multiplicity* is calculated with the function $F_{numSupplierConnects}$ (see Table 3.2). The *numConnect* attribute refers to the `ThresholdSpecification` named $Th_{maxConnects}$, (see Table 3.3), and it means that S_x might be a blob antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value. These two sub-views are related each other by a `BooleanRestriction` with the *OR type* that indicates that at least one of the two static sub-views must take place in the software architectural model to state a Blob antipattern occurrence.

The dynamic view is named *blob.dynamic* and it contains two `SubViews`. The first sub-view is named *blob.dynamic.controller*, it captures the BLOB-controller case in which a `SoftwareEntityInstance` S_x is involved in a `Service` as *sender* of

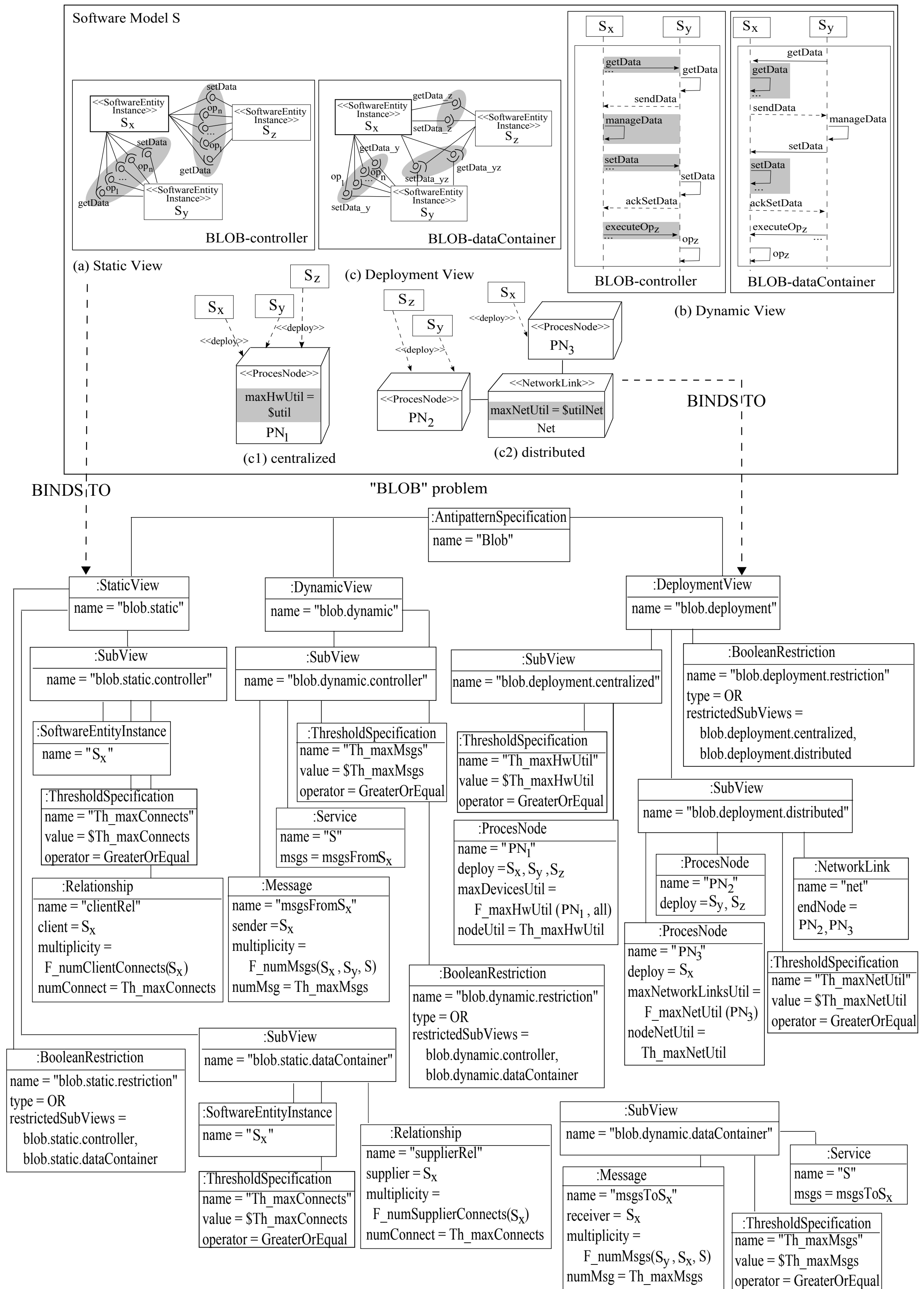


Figure 4.6: PAML-based model of the *Blob* Antipattern.

Messages, and the *multiplicity* is calculated with the function $F_{numMsgs}$ (see Table 3.2). The *numMsg* attribute refers to the ThresholdSpecification named $Th_{maxMsgs}$ (see Table 3.3), and it means that S_x might be a blob antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value. The second sub-view is *blob.dynamic.dataContainer*, it captures the BLOB-dataContainer case in which a SoftwareEntityInstance S_x is involved in a Service as *receiver* of Messages, and the *multiplicity* is calculated with the function $F_{numMsgs}$ (see Table 3.2). The *numMsg* attribute refers to the ThresholdSpecification named $Th_{maxMsgs}$, (see Table 3.3), and it means that S_x might be a blob antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value. These two sub-views are related each other by a BooleanRestriction with the *OR type* by indicating that at least one of the two dynamic sub-views must take place in the software architectural model to state a Blob antipattern occurrence.

The deployment view is named *blob.deployment* and it contains two SubViews. The first sub-view is named *blob.deployment.centralized*, it captures the centralized case in which the SoftwareEntityInstance S_x and the surrounding ones (e.g. s_y, s_z) are *deploy*ed on the ProcesNode named PN_1, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *nodeUtil* attribute refers to the ThresholdSpecification named $Th_{maxHwUtil}$ (see Table 3.4), and it means that S_x might be a blob antipattern instance if the *maxDevicesUtil* value is *GreaterOrEqual* to the threshold value. The second sub-view is *blob.deployment.distributed*, it captures the distributed case in which a SoftwareEntityInstance S_x and the surrounding ones (e.g. S_y, S_z) are *deploy*ed on different ProcesNodes named PN_2 and PN_3, and the *maxNetworkLinksUtil* is calculated with the function $F_{maxNetUtil}$ (see Table 3.2). The *nodeNetUtil* attribute refers to the ThresholdSpecification named $Th_{maxNetUtil}$, (see Table 3.4), and it means that S_x might be a blob antipattern instance if the *maxNetworkLinksUtil* value is *GreaterOrEqual* to the threshold value. These two sub-views are related each other by a BooleanRestriction with the *OR type* by indicating that at least one of the two deployment sub-views must take place in the software architectural model to state a Blob antipattern occurrence.

CONCURRENT PROCESSING SYSTEMS

Figure 4.7 shows the PAML-based model for the Concurrent Processing Systems antipattern. It contains one view: DeploymentView.

The deployment view is named *cps.deployment* and it contains three SubViews.

The first sub-view is named *cps.deployment.queue*, there are two ProcesNodes, i.e. PN_1 and PN_2, and PN_1 has the *maxQueueLength* calculated with the function F_{maxQL} (see Table 3.2). The *nodeQueueLength* attribute refers to the ThresholdSpecification named $Th_{maxQueue}$ (see Table 3.4), and it means that PN_1 might be a Concurrent Processing Systems antipattern instance if the *maxQueueLength* value is *GreaterOrEqual* to the threshold value. Additionally, PN_1 contains a

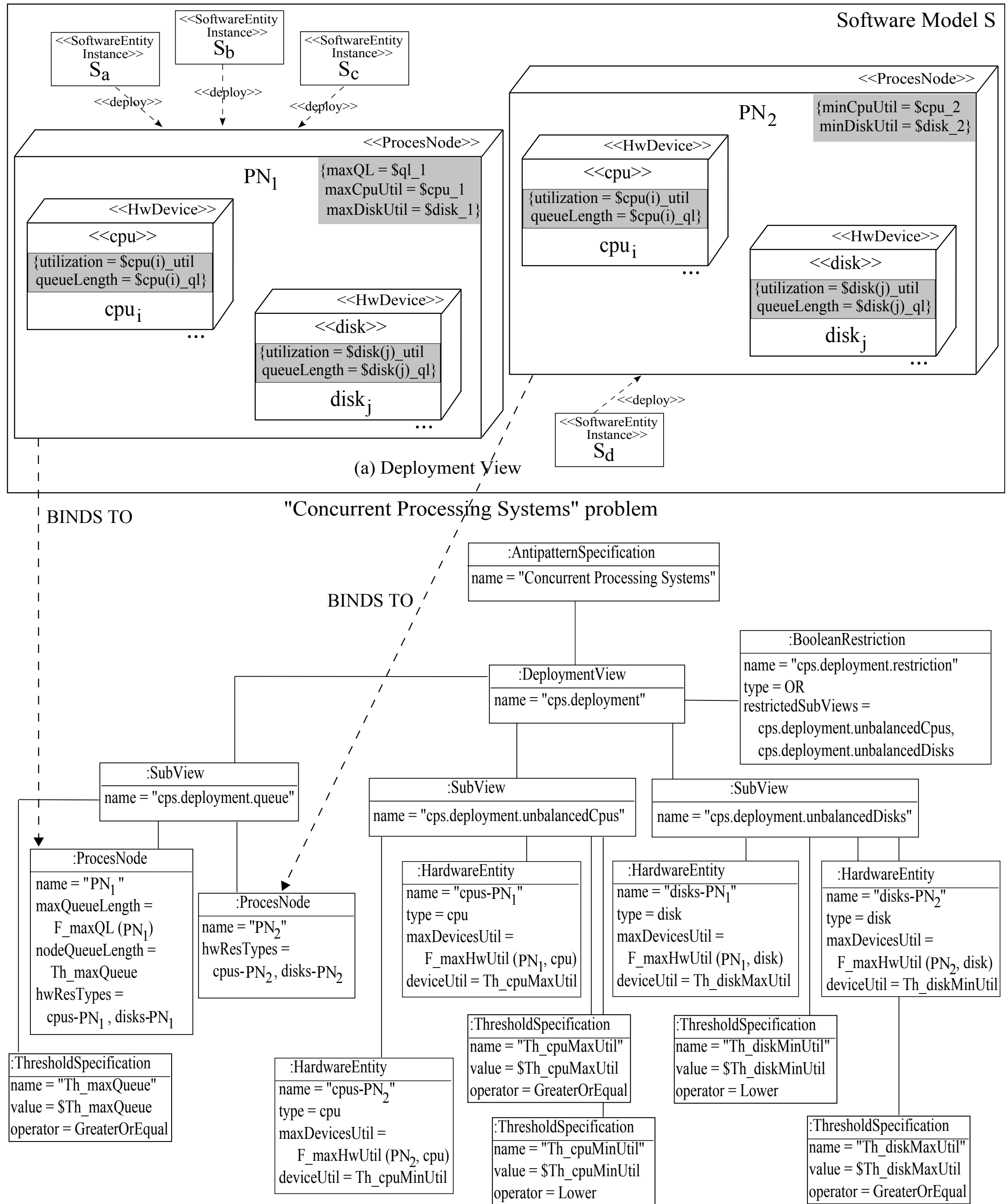


Figure 4.7: PAML-based model of the *Concurrent Processing Systems* Antipattern.

set of *hwResTypes* (i.e. $cpus - PN_1$, $disks - PN_1$), whereas PN_2 contains a set of *hwResTypes* (i.e. $cpus - PN_2$, $disks - PN_2$).

The second sub-view is *cps.deployment.unbalancedCpus*. The `HardwareEntity` named $cpus - PN_1$ collects all cpu devices of the processing node PN_1, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *deviceUtil* attribute refers to the `ThresholdSpecification` named $Th_{cpuMaxUtil}$ (see Table 3.4). The `HardwareEntity` named $cpus - PN_2$ collects all cpu devices of the processing node PN_2, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *deviceUtil* attribute refers to the `ThresholdSpecification` named $Th_{cpuMinUtil}$ (see Table 3.4). It means that PN_1 and PN_2 might be a Concurrent Processing Systems antipattern instance if the *maxDevicesUtil* value of $cpus - PN_1$ is *GreaterOrEqual* to the $Th_{cpuMaxUtil}$ threshold value, and the *maxDevicesUtil* value of $cpus - PN_2$ is *Lower* to the $Th_{cpuMinUtil}$ threshold value.

The third sub-view is *cps.deployment.unbalancedDisks*, and similarly to the the *cps.deployment.unbalancedCpus* checks the disks devices utilization: PN_1 and PN_2 might be a Concurrent Processing Systems antipattern instance if the *maxDevicesUtil* value of $disks - PN_1$ is *GreaterOrEqual* to the $Th_{diskMaxUtil}$ threshold value, and the *maxDevicesUtil* value of $disks - PN_2$ is *Lower* to the $Th_{diskMinUtil}$ threshold value. These sub-views *cps.deployment.unbalancedCpus* and *cps.deployment.unbalancedDisks* are related each other by a `BooleanRestriction` with the *OR type* by indicating that at least one of the two deployment sub-views must take place in the software architectural model to state a Concurrent Processing Systems antipattern occurrence.

PIPE AND FILTER ARCHITECTURES

Figure 4.8 shows the PAML-based model for the Pipe and Filter Architectures antipattern. It contains three views: `StaticView`, `DynamicView` and `DeploymentView`.

The static view is named *paf.static* and it contains one `SubView`, i.e. *paf.static.resDemands*. There is a `SoftwareEntityInstance` S_x that provides an `OperationInstance` op_x that has a *probability* to be executed *Equal* to 1. Such operation has a `StructuredResDemand` named *struct-RD-*op_x with three `BasicResDemands`: (i) *brd-comp-*op_x specifies the *computation* resource demand whose value is $\$comp - op_x$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{resDemand[comp]}$ (see Table 3.3) and it means that op_x might be a pipe and filter architectures antipattern instance if the *value* is *GreaterOrEqual* to the threshold value; (ii) *brd-stor-*op_x specifies the *storage* resource demand whose value is $\$stor - op_x$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{resDemand[stor]}$ (see Table 3.3) and it means that op_x might be a pipe and filter architectures antipattern instance if the *value* is *GreaterOrEqual* to the threshold value; (iii) *brd-band-*op_x specifies the *bandwidth* resource demand whose value is $\$band - op_x$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{resDemand[band]}$ (see Table 3.3) and it means

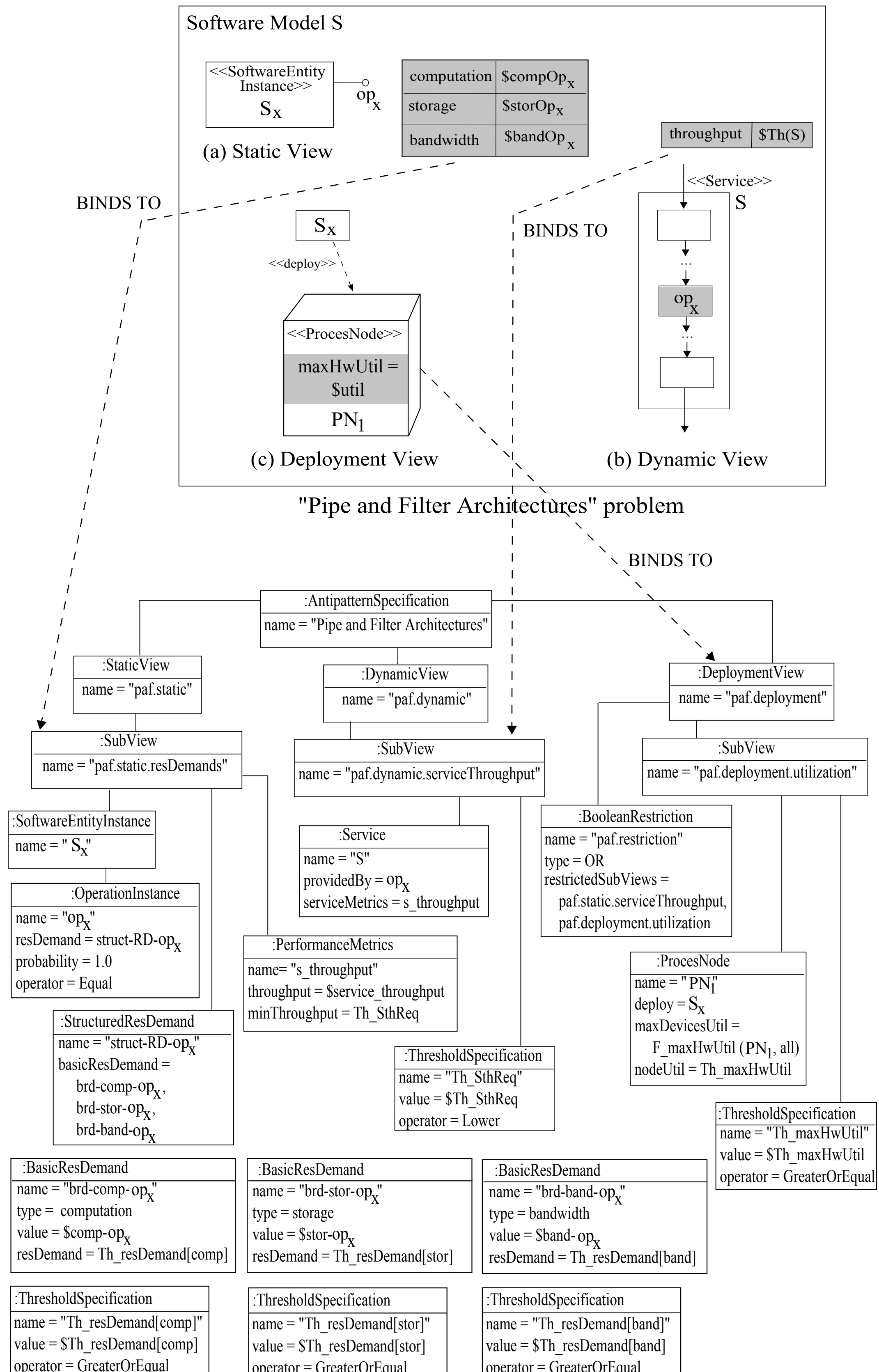


Figure 4.8: PAML-based model of the *Pipe and Filter Architectures* Antipattern.

that op_x might be a pipe and filter architectures antipattern instance if the *value* is *GreaterOrEqual* to the threshold value.

The dynamic view is named *paf.dynamic* and it contains one `SubView`, i.e. *paf.dynamic.serviceThroughput*. There is a `Service` S *providedBy* the operation instance op_x that has the *serviceMetrics* referring to the `PerformanceMetrics` named *s_throughput*. The numerical value of the service throughput is $\$service_throughput$, and the *minThroughput* attribute refers to the `ThresholdSpecification` named Th_{SthReq} (see Table 3.5) and it means that op_x might be a pipe and filter architectures antipattern instance if the *throughput* is *Lower* than the threshold value.

The deployment view is named *paf.deployment* and it contains one `SubView`, i.e. *paf.deployment.utilization*. There is a `ProcesNode` PN_1 on which S_x is deployed, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *nodeUtil* attribute refers to the `ThresholdSpecification` named $Th_{maxHwUtil}$ (see Table 3.4), and it means that S_x might be a pipe and filter architectures antipattern instance if the *maxDevicesUtil* value is *GreaterOrEqual* to the threshold value. The sub-views *paf.dynamic.serviceThroughput* and *paf.deployment.utilization* are related each other by a `BooleanRestriction` with the *OR type* by indicating that at least one of the two sub-views must take place in the software architectural model to state a Pipe and Filter Architectures antipattern occurrence.

EXTENSIVE PROCESSING

Figure 4.9 shows the PAML-based model for the Extensive Processing antipattern. It contains three views: `StaticView`, `DynamicView` and `DeploymentView`.

The static view is named *ep.static* and it contains one `SubView`, i.e. *ep.static.resDemands*. There is a `SoftwareEntityInstance` S_x that provides two `OperationInstances`, i.e. op_x and op_y. Both have a *probability* to be executed *Lower* than 1. The op_x operation has a `StructuredResDemand` named *struct-RD-*op_x with three `BasicResDemands`: (i) *brd-comp-*op_x specifies the *computation* resource demand whose value is $\$comp - op_x$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{maxResDemand[comp]}$ (see Table 3.3) and it means that op_x might be an extensive processing antipattern instance if the *value* is *GreaterOrEqual* to the threshold value; (ii) *brd-stor-*op_x specifies the *storage* resource demand whose value is $\$stor - op_x$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{maxResDemand[stor]}$ (see Table 3.3) and it means that op_x might be an extensive processing antipattern instance if the *value* is *GreaterOrEqual* to the threshold value; (iii) *brd-band-*op_x specifies the *bandwidth* resource demand whose value is $\$band - op_x$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{maxResDemand[band]}$ (see Table 3.3) and it means that op_x might be an extensive processing antipattern instance if the *value* is *GreaterOrEqual* to the threshold value. The op_y operation has a `StructuredResDemand` named *struct-RD-*op_y with three `BasicResDemands`: (i)

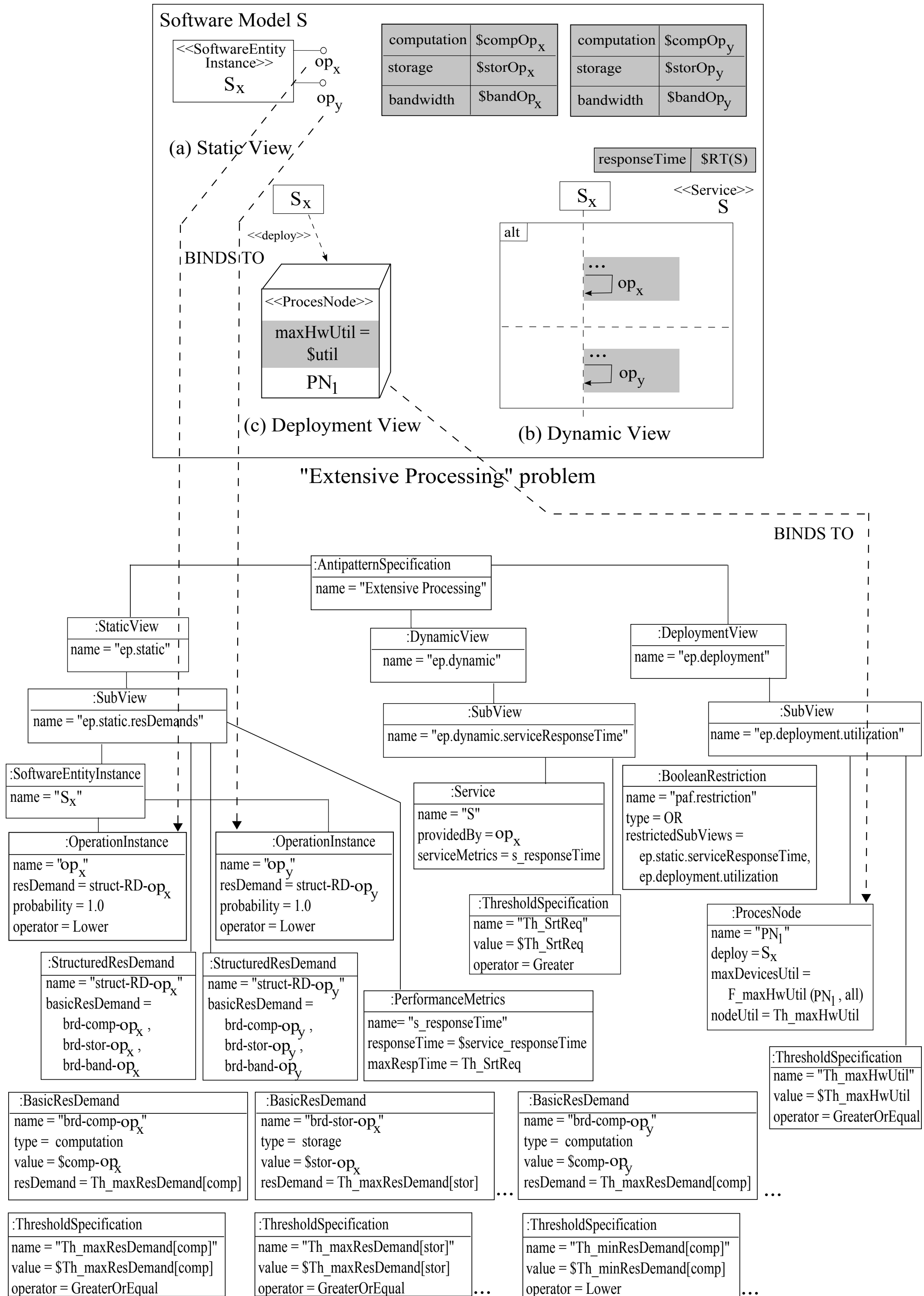


Figure 4.9: PAML-based model of the *Extensive Processing* Antipattern.

brd-comp-op_y specifies the *computation* resource demand whose value is $\$comp - op_y$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{minResDemand[comp]}$ (see Table 3.3) and it means that op_y might be an extensive processing antipattern instance if the *value* is *Lower* than the threshold value; (ii) *brd-stor-*op_y specifies the *storage* resource demand whose value is $\$stor - op_y$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{minResDemand[stor]}$ (see Table 3.3) and it means that op_y might be an extensive processing antipattern instance if the *value* is *Lower* than the threshold value; (iii) *brd-band-*op_y specifies the *bandwidth* resource demand whose value is $\$band - op_y$, the *resDemand* attribute refers to the `ThresholdSpecification` named $Th_{minResDemand[band]}$ (see Table 3.3) and it means that op_y might be an extensive processing antipattern instance if the *value* is *Lower* than the threshold value.

The dynamic view is named *ep.dynamic* and it contains one `SubView`, i.e. *ep.dynamic.serviceResponseTime*. There is a `Service` S *providedBy* the operation instance op_x that has the *serviceMetrics* referring to the `PerformanceMetrics` named *s_responseTime*. The numerical value of the service response time is $\$service_responseTime$, and the *maxRespTime* attribute refers to the `ThresholdSpecification` named Th_{SrtReq} (see Table 3.5) and it means that op_x might be an extensive processing antipattern instance if the *responseTime* is *Greater* than the threshold value.

The deployment view is named *ep.deployment* and it contains one `SubView`, i.e. *ep.deployment.utilization*. There is a `ProcesNode` PN_1 on which S_x is deployed, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *nodeUtil* attribute refers to the `ThresholdSpecification` named $Th_{maxHwUtil}$ (see Table 3.4), and it means that S_x might be an extensive processing antipattern instance if the *maxDevicesUtil* value is *GreaterOrEqual* to the threshold value. The sub-views *paf.dynamic.serviceThroughput* and *paf.deployment.utilization* are related each other by a `BooleanRestriction` with the *OR type* by indicating that at least one of the two sub-views must take place in the software architectural model to state a Pipe and Filter Architectures antipattern occurrence.

CIRCUITOUS TREASURE HUNT

Figure 4.10 shows the PAML-based model for the Circuitous Treasure Hunt antipattern. It contains three views: `StaticView`, `DynamicView` and `DeploymentView`.

The static view is named *cth.static* and it contains the `SubView` named *cth.static.swRes* with two `SoftwareEntityInstances`, i.e. S_x and $Database$ that is a database, as stated by the boolean attribute *isDB*.

The dynamic view is named *cth.dynamic* and it contains the `SubView` *cth.dynamic.msgTraffic* in which the `SoftwareEntityInstances` S_x and $Database$ are involved in a `Service` respectively as *sender* and *receiver* of `Messages`, and the

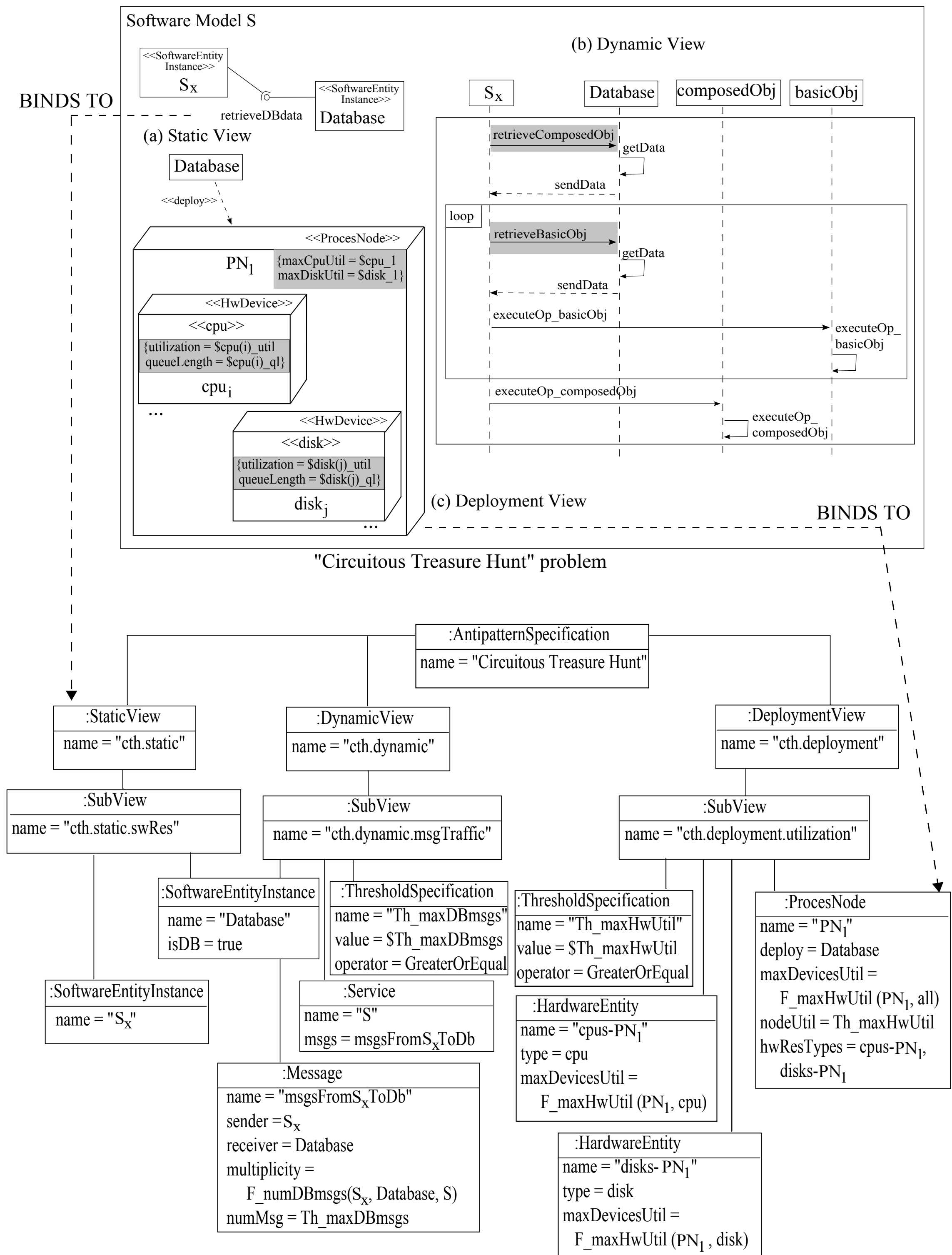


Figure 4.10: PAML-based model of the *Circuitous Treasure Hunt* Antipattern.

multiplicity is calculated with the function $F_{numDBMsgs}$ (see Table 3.2). The *numMsg* attribute refers to the ThresholdSpecification named $Th_{maxDBmsgs}$ (see Table 3.3), and it means that $Database$ might be a circuitous treasure hunt antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value.

The deployment view is named *cth.deployment* and it contains the SubView named *cth.deployment.utilization* in which the SoftwareEntityInstance $Database$ is *deploy*ed on the ProcesNode named PN_1, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *nodeUtil* attribute refers to the ThresholdSpecification named $Th_{maxHwUtil}$ (see Table 3.4), and it means that $Database$ might be a circuitous treasure hunt antipattern instance if the *maxDevicesUtil* value is *GreaterOrEqual* to the threshold value. Additionally, PN_1 contains a set of *hwResTypes* (i.e. $cpus - PN_1$, $disks - PN_1$). The HardwareEntity named $cpus - PN_1$ collects all cpu devices of the processing node PN_1, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The HardwareEntity named $disks - PN_1$ collects all disk devices of the processing node PN_1, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). It means that $Database$ might be a Circuitous Treasure Hunt antipattern instance if the $disks - PN_1$ is more used than $cpus - PN_1$, i.e. the *maxDevicesUtil* value of $disks - PN_1$ is greater than the *maxDevicesUtil* value of $cpus - PN_1$.

EMPTY SEMI TRUCKS

Figure 4.11 shows the PAML-based model for the Empty Semi Trucks antipattern. It contains three views: StaticView, DynamicView and DeploymentView.

The static view is named *est.static* and it contains the SubView named *est.static.swRes* with one SoftwareEntityInstance S_x.

The dynamic view is named *est.dynamic* and it contains the SubView *est.dynamic.remMsgs* in which the SoftwareEntityInstance S_x is involved in a Service as *sender* of Messages, and the *numRemMsgs* is calculated with the function $F_{numRemMsgs}$ (see Table 3.2). The *remMsgs* attribute refers to the ThresholdSpecification named $Th_{maxRemMsgs}$ (see Table 3.3), and it means that S_x might be an empty semi trucks antipattern instance if the *numRemMsgs* value is *GreaterOrEqual* to the threshold value. The *numRemInstances* is calculated with the function $F_{numRemInst}$ (see Table 3.2), and the *remInstances* attribute refers to the ThresholdSpecification named $Th_{maxRemInst}$ (see Table 3.3), and it means that S_x might be an empty semi trucks antipattern instance if the *numRemInstances* value is *GreaterOrEqual* to the threshold value.

The deployment view is named *est.deployment* and it contains one SubView named *est.deployment.utilization*. The sub-view contains the ProcesNode named PN_1 in which the SoftwareEntityInstance S_x is *deploy*ed; the *maxDevicesUtil* and *maxNetworkLinksUtil* attributes are calculated with the functions $F_{maxHwUtil}$

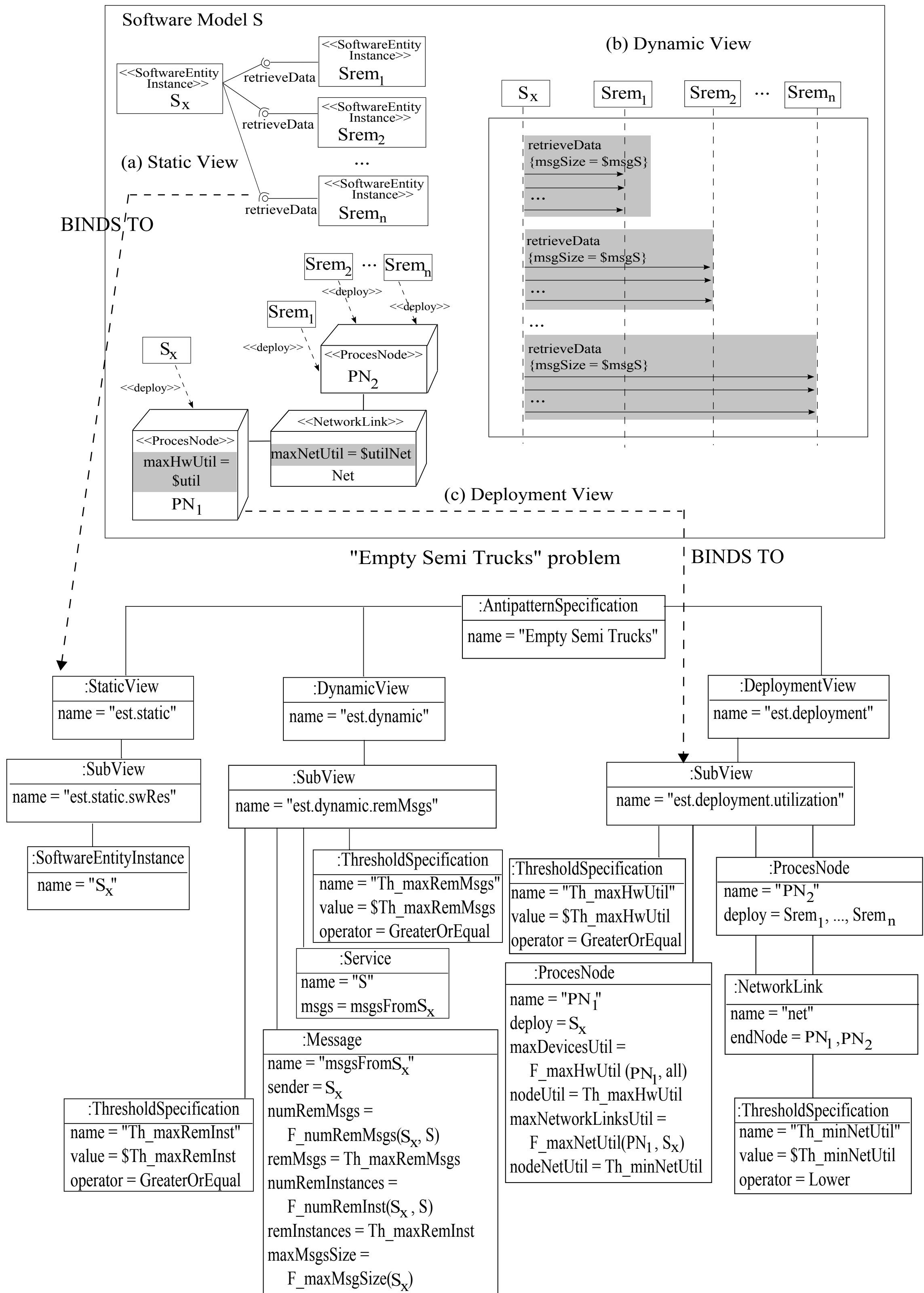


Figure 4.11: PAML-based model of the *Empty Semi Trucks* Antipattern.

and $F_{maxNetUtil}$ respectively (see Table 3.2). The *nodeUtil* attribute refers to the `ThresholdSpecification` named $Th_{maxHwUtil}$ (see Table 3.4), and it means that S_x might be an empty semi trucks antipattern instance if the *maxDevicesUtil* value is *GreaterOrEqual* to the threshold value. The *nodeNetUtil* attribute refers to the `ThresholdSpecification` named $Th_{minNetUtil}$ (see Table 3.4), and it means that S_x might be an empty semi trucks antipattern instance if the *maxNetworkLinksUtil* value is *Lower* to the threshold value.

TOWER OF BABEL

Figure 4.12 shows the PAML-based model for the Tower of Babel antipattern. It contains three views: `StaticView`, `DynamicView` and `DeploymentView`.

The static view is named *tob.static* and it contains the `SubView` named *tob.static.swRes* with one `SoftwareEntityInstance` S_x, and one `OperationInstance` op_x.

The dynamic view is named *tob.dynamic* and it contains the `SubView` *tob.dynamic.translateMsgs* in which the `SoftwareEntityInstance` S_x is involved in a `Service` since it is *providedBy* op_x. The *exchangeFormats* is calculated with the function F_{numExF} (see Table 3.2), and the *numFormats* attribute refers to the `ThresholdSpecification` named Th_{maxExF} (see Table 3.3), and it means that S_x might be a tower of babel antipattern instance if the *exchangeFormats* value is *GreaterOrEqual* to the threshold value.

The deployment view is named *tob.deployment* and it contains the `SubView` named *tob.deployment.utilization*. There is the `ProcesNode` named PN_1 in which the `SoftwareEntityInstance` S_x is *deploy*ed, and the *maxDevicesUtil* is calculated with the function $F_{maxHwUtil}$ (see Table 3.2). The *nodeUtil* attribute refers to the `ThresholdSpecification` named $Th_{maxHwUtil}$ (see Table 3.4), and it means that S_x might be a tower of babel antipattern instance if the *maxDevicesUtil* value is *GreaterOrEqual* to the threshold value.

ONE-LANE BRIDGE

Figure 4.13 shows the PAML-based model for the One-Lane Bridge antipattern. It contains three views: `StaticView`, `DynamicView` and `DeploymentView`.

The static view is named *olb.static* with the `SubView` named *olb.static.swResCapacity* containing one `SoftwareEntityInstance` S_x whose *capacity* is calculated with the function $F_{poolSize}$ (see Table 3.2), and one `OperationInstance` op_x.

The dynamic view is named *olb.dynamic* and it contains two `SubViews`, i.e. *olb.dynamic.synchCalls* and *olb.dynamic.serviceResponseTime*. The first sub-view contains the `SoftwareEntityInstance` S_x that is involved in a `Service` since it is

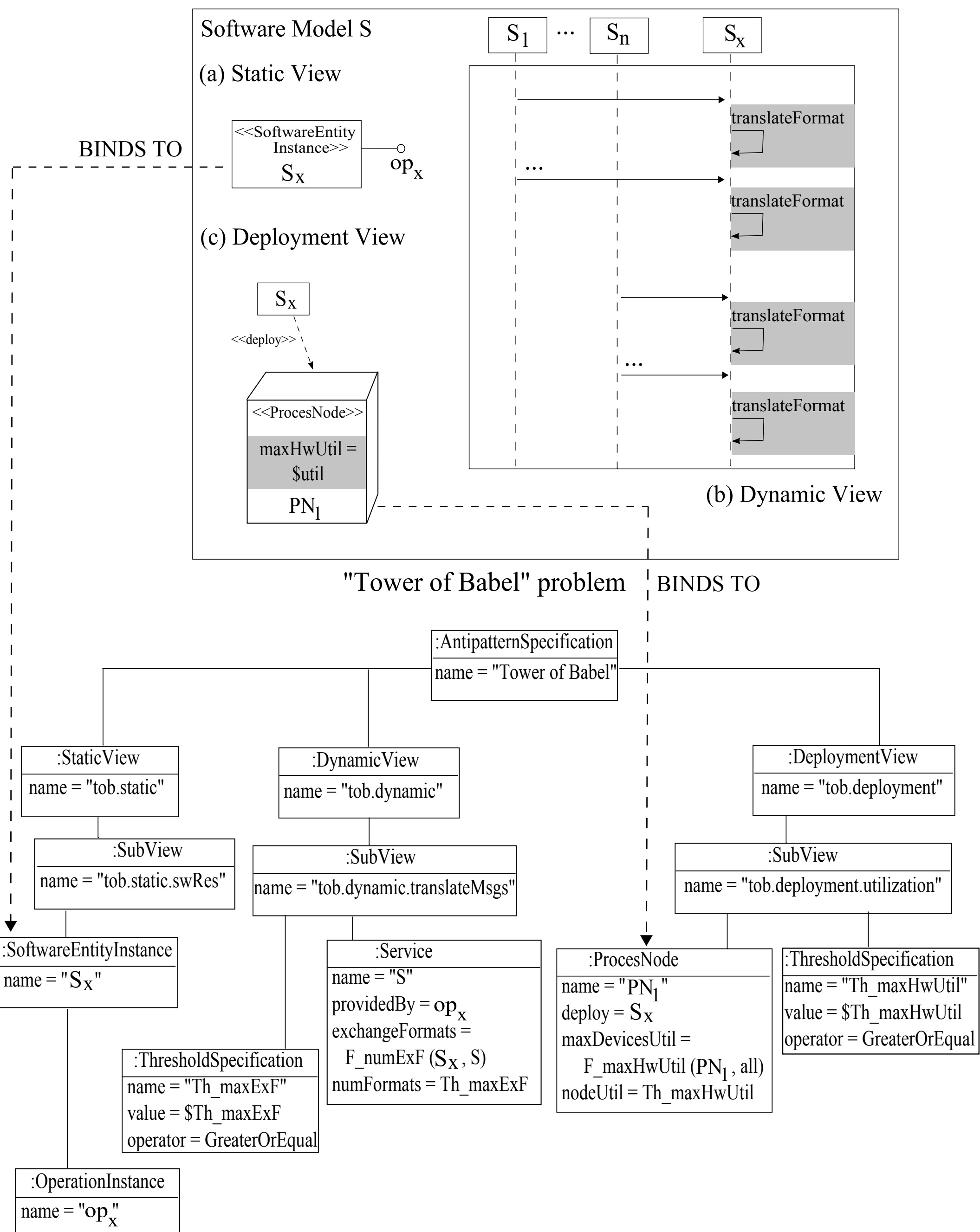


Figure 4.12: PAML-based model of the *Tower of Babel* Antipattern.

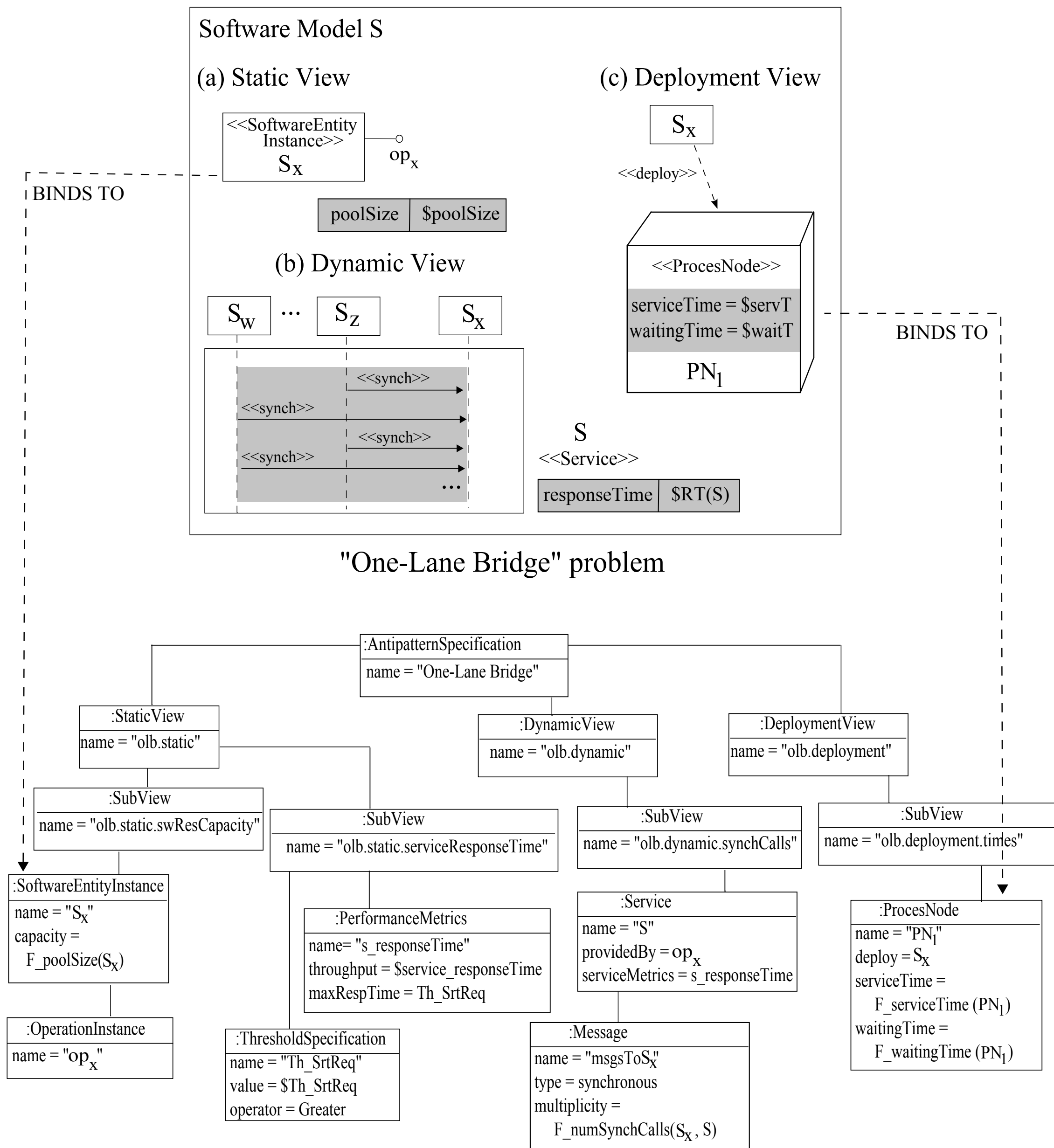


Figure 4.13: PAML-based model of the *One-Lane Bridge* Antipattern.

providedBy op_x. The software entity instance S_x is involved as *receiver* of `Messages`, and the *multiplicity* is calculated with the function $F_{numSynchCalls}$ (see Table 3.2), and it will be compared with the *capacity* of S_x.

The second sub-view states that the `Service` S has the *serviceMetrics* referring to the `PerformanceMetrics` named *s_responseTime*. The numerical value of the service response time is $\$service_responseTime$, and the *maxRespTime* attribute refers to the `ThresholdSpecification` named Th_{SrtReq} (see Table 3.5) and it means that S_x might be a one-lane bridge antipattern instance if the *responseTime* is *Greater* than the threshold value.

The deployment view is named *olb.deployment* and it contains the `SubView` named *olb.deployment.times*. There is the `ProcesNode` named PN_1 in which the `SoftwareEntityInstance` S_x is *deploy*ed. Two performance indices are calculated and compared: (i) the *serviceTime* is calculated with the function $F_{serviceTime}$ (see Table 3.2); (ii) the *waitingTime* is calculated with the function $F_{waitingTime}$ (see Table 3.2). It means that S_x might be a one-lane bridge antipattern instance if the *serviceTime* value is *Lower* than the *waitingTime*.

EXCESSIVE DYNAMIC ALLOCATION

Figure 4.14 shows the PAML-based model for the Excessive Dynamic Allocation antipattern. It contains two views: `StaticView` and `DynamicView`.

The static view is named *eda.static*, it contains the `SubView` named *eda.static.swRes* with one `SoftwareEntityInstance` S_x and one `OperationInstance` op_x.

The dynamic view is named *eda.dynamic* and it contains three `SubViews`.

The first sub-view is named *eda.dynamic.serviceResponseTime*, and the `SoftwareEntityInstance` S_x is involved in a `Service` since it is *providedBy* op_x. The service S has the *serviceMetrics* referring to the `PerformanceMetrics` named *s_responseTime*. The numerical value of the service response time is $\$service_responseTime$, and the *maxRespTime* attribute refers to the `ThresholdSpecification` named Th_{SrtReq} (see Table 3.5) and it means that S_x might be an excessive dynamic allocation antipattern instance if the *responseTime* is *Greater* than the threshold value.

The second sub-view is named *eda.dynamic.createObj*, and the software entity instance S_x is involved as *sender* of `Messages` whose semantic is explicitly stated by the boolean attribute *isCreateObjectAction*. The *multiplicity* is calculated with the function $F_{numCreatedObj}$ (see Table 3.2), and the *numMsg* attribute refers to the `ThresholdSpecification` named $Th_{maxCrObj}$ (see Table 3.3), and it means that S_x might be an excessive dynamic allocation antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value.

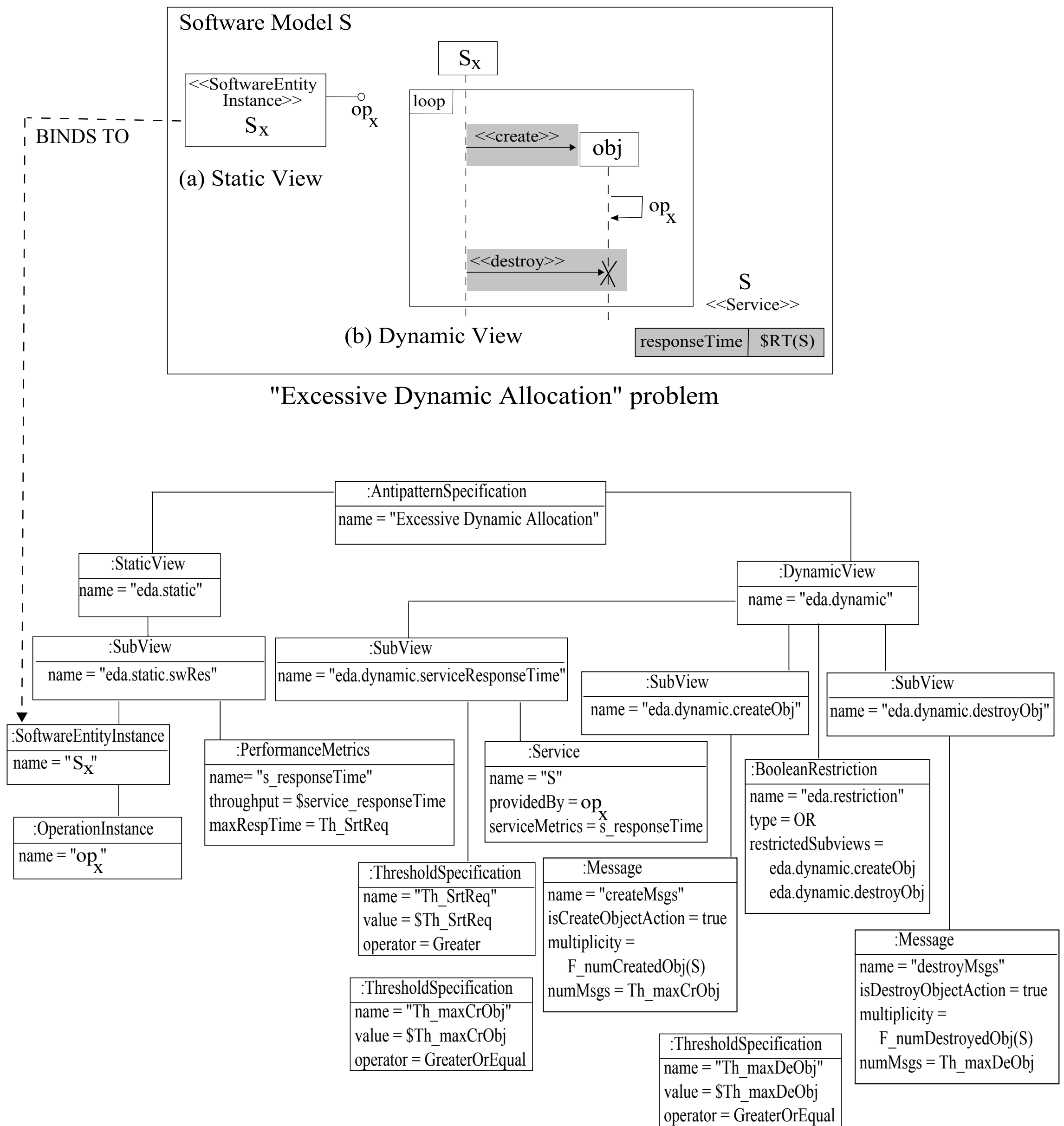


Figure 4.14: PAML-based model of the *Excessive Dynamic Allocation* Antipattern.

The third sub-view is named *eda.dynamic.destroyObj*, and the software entity instance S_x is involved as *sender* of `Messages` whose semantic is explicitly stated by the boolean attribute *isDestroyObjectAction*. The *multiplicity* is calculated with the function $F_{numDestroyedObj}$ (see Table 3.2), and the *numMsg* attribute refers to the `ThresholdSpecification` named $Th_{maxDeObj}$ (see Table 3.3), and it means that S_x might be an excessive dynamic allocation antipattern instance if the *multiplicity* value is *GreaterOrEqual* to the threshold value. The sub-views *eda.dynamic.createObj* and *eda.dynamic.destroyObj* are related each other by a `BooleanRestriction` with the *OR type* by indicating that at least one of the two sub-views must take place in the software architectural model to state an Excessive Dynamic Allocation antipattern occurrence.

4.2.2 MULTIPLE-VALUES PERFORMANCE ANTIPATTERNS

In this Section we report the *Performance Antipatterns* that to be detected require the trend (or evolution) of the performance indices along the time (i.e. multiple-values). The numerical values of the performance indices (e.g. the operation response time) come from the simulation of the performance model.

TRAFFIC JAM

Figure 4.15 shows the PAML-based model for the Traffic Jam antipattern. It contains one view: `StaticView`.

The static view is named *tj.static*, it contains the `SubView` named *tj.static.respTime* with one `SoftwareEntityInstance` S_x and one `OperationInstance` op_x. The operation instance op_x has the *opMetrics* referring to the `PerformanceMetrics` named op_x*-responseTime*. The numerical values of the operation response time are calculated with the function $F_{RT}(op_x, t_i)$ (see Table 3.2), where t_i specifies the interval of time after which the performance index is evaluated. The *slope* attribute refers to the `ThresholdSpecification` named Th_{rtVar} (see Table 3.6) and it means that op_x might be a traffic jam antipattern instance if the *responseTime* is *Greater* than the threshold value.

THE RAMP

Figure 4.16 shows the PAML-based model for The Ramp antipattern. It contains one view: `StaticView`.

The static view is named *tr.static*, it contains two `SubViews` named *tr.static.respTime* and *tr.static.throughput*. Similarly to the Traffic Jam antipattern, the first sub-view contains one `SoftwareEntityInstance` S_x and one `OperationInstance` op_x. The operation instance op_x has the *opMetrics* referring to the `PerformanceMetrics`

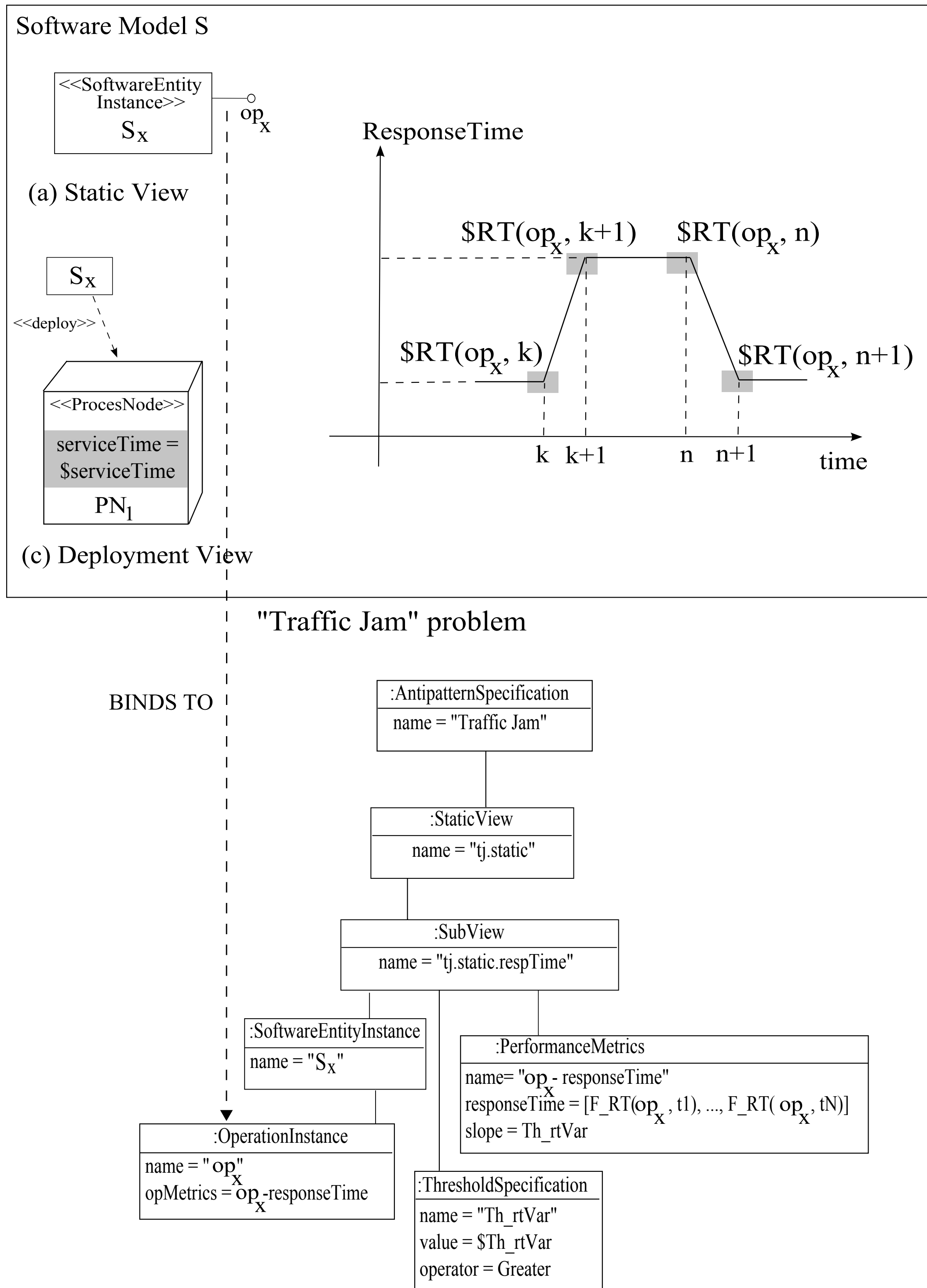


Figure 4.15: PAML-based model of the *Traffic Jam* Antipattern.

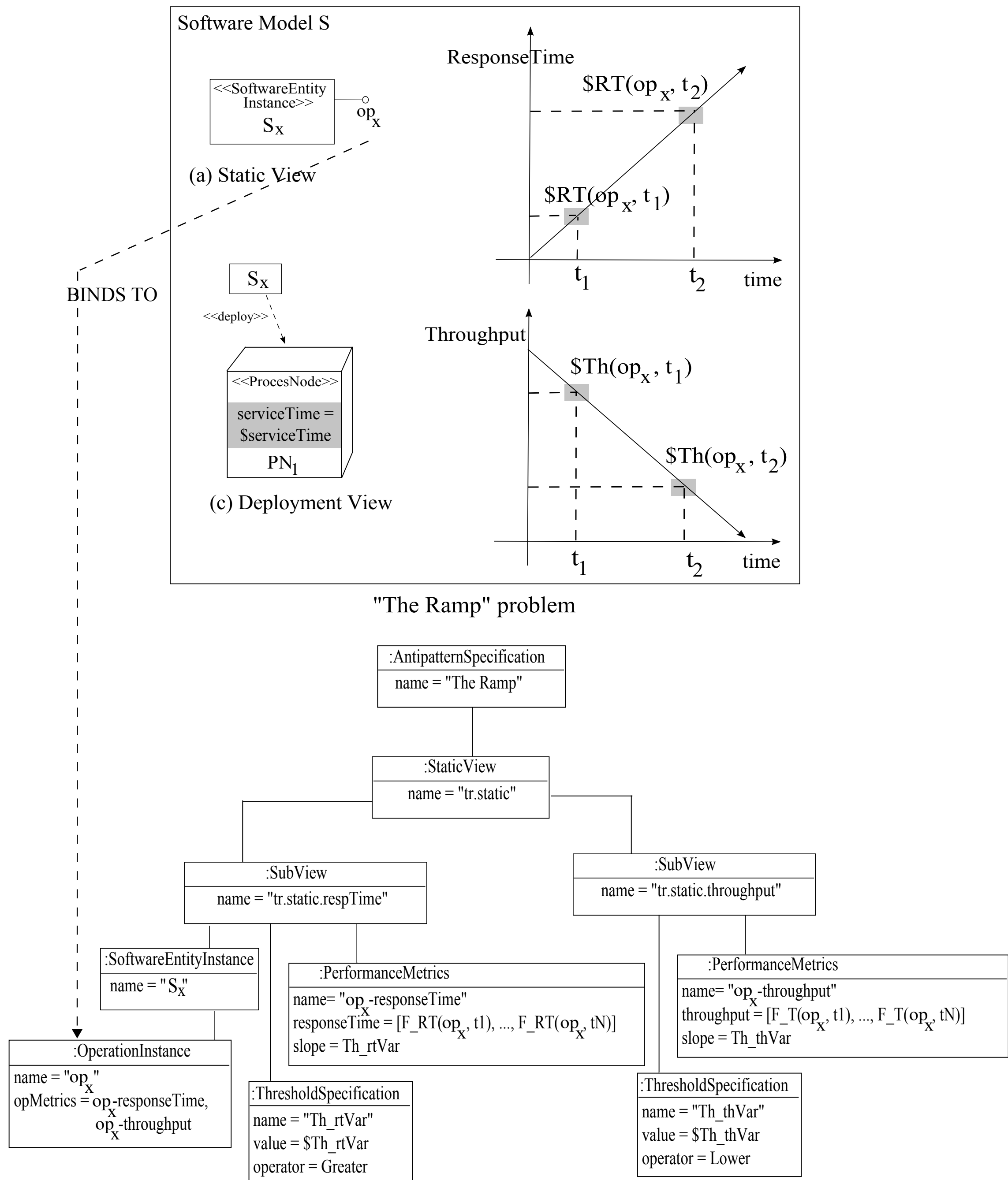


Figure 4.16: PAML-based model of the *The Ramp* Antipattern.

named op_x-*responseTime* and op_x-*throughput*. The numerical values of the operation response time and throughput are calculated respectively with the function $F_{RT}(op_x, t_i)$ and $F_T(op_x, t_i)$ (see Table 3.2), where t_i specifies the interval of time after which the performance indices are evaluated. In case of the response time, the *slope* attribute refers to the `ThresholdSpecification` named Th_{rtVar} (see Table 3.6) and it means that op_x might be the ramp antipattern instance if the *responseTime* is *Greater* than the threshold value. In case of the throughput, the *slope* attribute refers to the `ThresholdSpecification` named Th_{thVar} (see Table 3.6) and it means that op_x might be the ramp antipattern instance if the *throughput* is *Lower* than the threshold value.

MORE IS LESS

Figure 4.17 shows the PAML-based model for the More Is Less antipattern. It contains one view: `DeploymentView`.

The deployment view is named *mil.deployment*, it contains one `SubView` named *mil.deployment.params* in which one `ProcesNode` PN_1 has the *runTimeParams* referring to the `Params` named PN_1-*RTparams*. The numerical values of the run time parameters are calculated respectively with the function $F_{par}(PN_1, pType, t_i)$ (see Table 3.2), where t_i specifies the interval of time after which the performance indices are evaluated. The *slope* attribute refers to the `ThresholdSpecification` named $Th_{maxParams}$ (see Table 3.3) and it means that PN_1 might be a more is less antipattern instance if the *runTimeParams* are *Greater* than the threshold values.

4.2.3 SUMMARY

This Section summarizes the *metamodel* elements introduced for representing antipatterns as PAML-based models. Table 4.1 lists the metamodel-based representation of the performance antipatterns we propose. Each row represents a specific antipattern that is characterized by four attributes: *antipattern* name, the *Static View*, *Dynamic View*, *Deployment View* metamodel elements the corresponding antipattern requires.

Note that the *ThresholdSpecification* element introduces a degree of uncertainty. We are working on defining a metric that quantifies such uncertainty as a function of the number of the thresholds an antipattern formalization requires. Just to give a hint, from Table 4.1 we can notice that the Blob antipattern is built on the metamodel elements belonging to all the three views we consider, and each view contains a *ThresholdSpecification* element; on the contrary, the One-Lane Bridge antipattern is also built on the metamodel elements belonging to all the three views we consider, but only the Dynamic View contains a *ThresholdSpecification* element. This observation leads us to guess that the detection of the Blob antipattern might have more elements of fuzziness in comparison to the detection of the One-Lane Bridge antipattern.

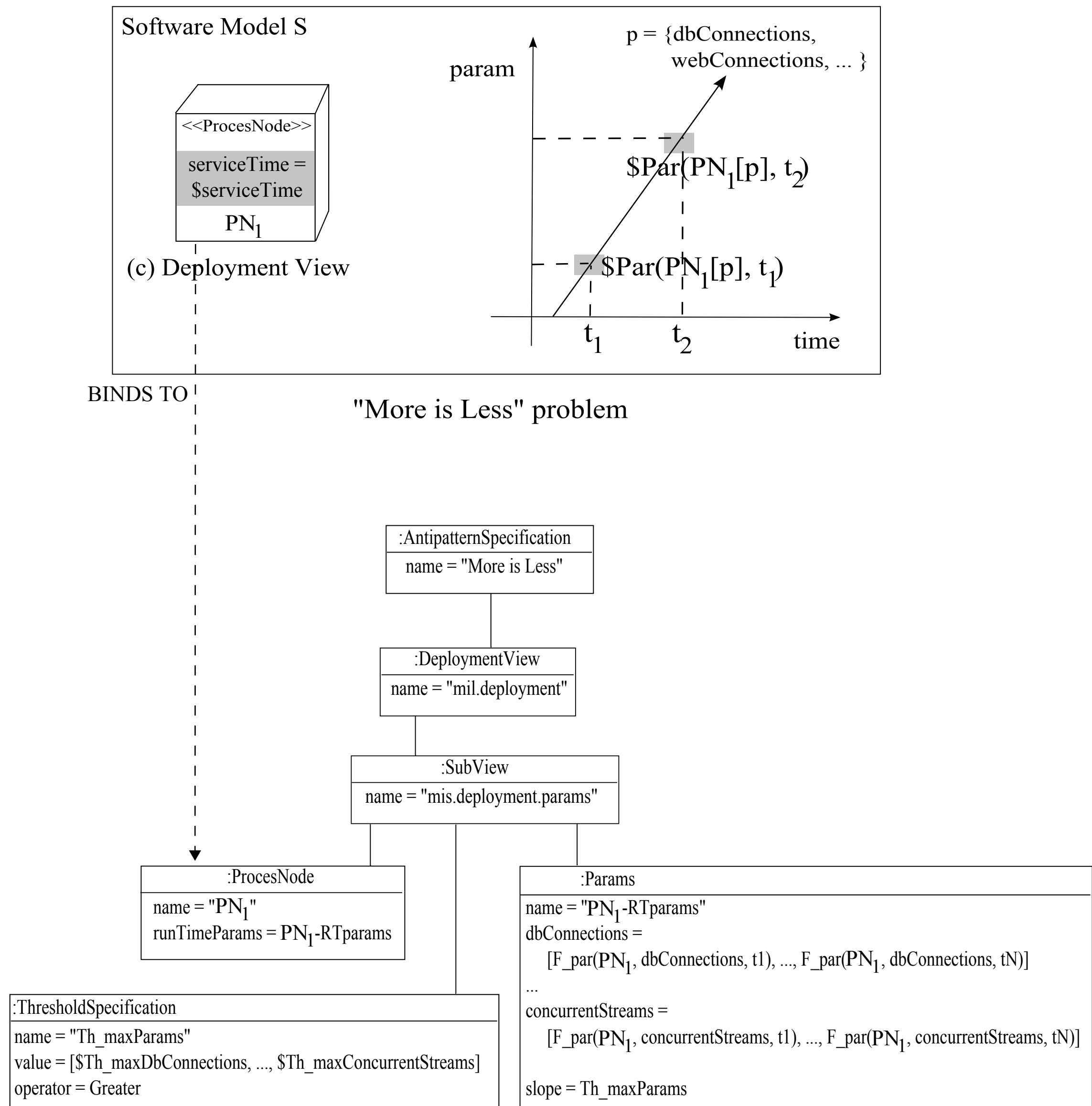


Figure 4.17: PAML-based model of the *More Is Less* Antipattern.

	Antipattern		Static View	Dynamic View	Deployment View
Single-value	Blob (or god class/component)		*SoftwareEntityInstance*, *Relationship*, *ThresholdSpecification*	*Service*, *Message*, *ThresholdSpecification*	*ProcesNode*, *NetworkLink*, *ThresholdSpecification*
	Unbalanced Processing	Concurrent Processing Systems	-	-	*ProcesNode*, *HardwareEntity*, *ThresholdSpecification*
		“Pipe and Filter” Architectures	*SoftwareEntityInstance*, *OperationInstance*, *StructuredResDemand*, *BasicResDemand*, *PerformanceMetrics*, *ThresholdSpecification*	*Service*, *ThresholdSpecification*	*ProcesNode*, *ThresholdSpecification*
		Extensive Processing	*SoftwareEntityInstance*, *OperationInstance*, *StructuredResDemand*, *BasicResDemand*, *PerformanceMetrics*, *ThresholdSpecification*	*Service*, *ThresholdSpecification*	*ProcesNode*, *ThresholdSpecification*
	Circuitous Treasure Hunt		*SoftwareEntityInstance*	*Service*, *Message*, *ThresholdSpecification*	*ProcesNode*, *HardwareEntity*, *ThresholdSpecification*
	Empty Semi Trucks		*SoftwareEntityInstance*	*Service*, *Message*, *ThresholdSpecification*	*ProcesNode*, *NetworkLink*, *ThresholdSpecification*
	Tower of Babel		*SoftwareEntityInstance*	*Service*, *ThresholdSpecification*	*ProcesNode*, *ThresholdSpecification*
	One-Lane Bridge		*SoftwareEntityInstance*, *OperationInstance*, *PerformanceMetrics*	*Service*, *Message*, *ThresholdSpecification*	*ProcesNode*
	Excessive Dynamic Allocation		*SoftwareEntityInstance*, *OperationInstance*, *PerformanceMetrics*	*Service*, *Message*, *ThresholdSpecification*	-
Multiple-values	Traffic Jam		*SoftwareEntityInstance*, *OperationInstance*, *PerformanceMetrics*, *ThresholdSpecification*	-	-
	The Ramp		*SoftwareEntityInstance*, *OperationInstance*, *PerformanceMetrics*, *ThresholdSpecification*	-	-
	More is Less		-	-	*ProcesNode*, *Params*, *ThresholdSpecification*

Table 4.1: A metamodel-based representation of Performance Antipatterns.

4.3 Towards the specification of a model-based framework

The benefit of introducing a metamodel approach for specifying antipatterns is manifold: (i) *expressiveness*, as it currently contains all the concepts needed to specify performance antipatterns introduced in [123]; (ii) *usability*, as it allows a user-friendly representation of (existing and upcoming) performance antipatterns; (iii) *extensibility*, i.e. if new antipatterns are based on additional concepts the metamodel can be extended to introduce such concepts.

Note that the subset of the antipattern types can be enlarged as far as the concepts for representing such types are available. Technology-specific antipatterns such as EJB and J2EE antipatterns [52] [127] can be also suited to check if the current metamodel is reusable in domain-specific fields. For example, we retain that the EJB Bloated Session Bean Antipattern [52] can be currently specified as a PAML-based model, since it describes a situation in EJB systems where a session bean has become too bulky, thus it is very similar to the Blob antipattern in the Smith-Williams' classification.

Performance antipatterns are built on a set of model elements belonging to SML+, i.e. a neutral notation agnostic of any concrete modeling language. For example, the *Blob* antipattern specification (see Figure 4.18) contains the *SoftwareEntity* and *ProcesNode* model elements, whereas it is not related to the *StructuredResDemand* element.

Figure 4.18 additionally shows how the neutral specification of performance antipatterns can be translated into concrete modeling languages. In fact SML+ is meant to provide the infrastructure upon which constructing the semantic relations among different notations.

This thesis considers two notations: UML [12] and Marte profile[2] [13]; the Palladio Component Model (PCM) [22]. Note that the subset of target modeling languages can be enlarged as far as the concepts for representing antipatterns are available; for example, architectural languages such as Æmilia [26] (see more details in Appendix B.2) can be also suited to validate the approach.

The semantic relations with SML+ depend on the expressiveness of the target modeling language. For example, in Figure 4.18 we can notice that a *SoftwareEntity* is respectively translated in a *UML Component*, a *PCM Basic Component*, and an *Æmilia ARCHI_ELEM_TYPE*. On the contrary, a full mapping is not possible for a *ProcesNode* whose translation is only possible to a *UML Node* and a *PCM Resource Container*, whereas in Æmilia this concept remains uncovered.

We can therefore assert that in a concrete modeling language there are antipatterns that can be automatically detected (i.e. when the entire set of SML+ model elements can be translated in the concrete modeling language) and some others that are no detectable (i.e.

[2]MARTE profile provides facilities to annotate UML models with information required to perform performance analysis.

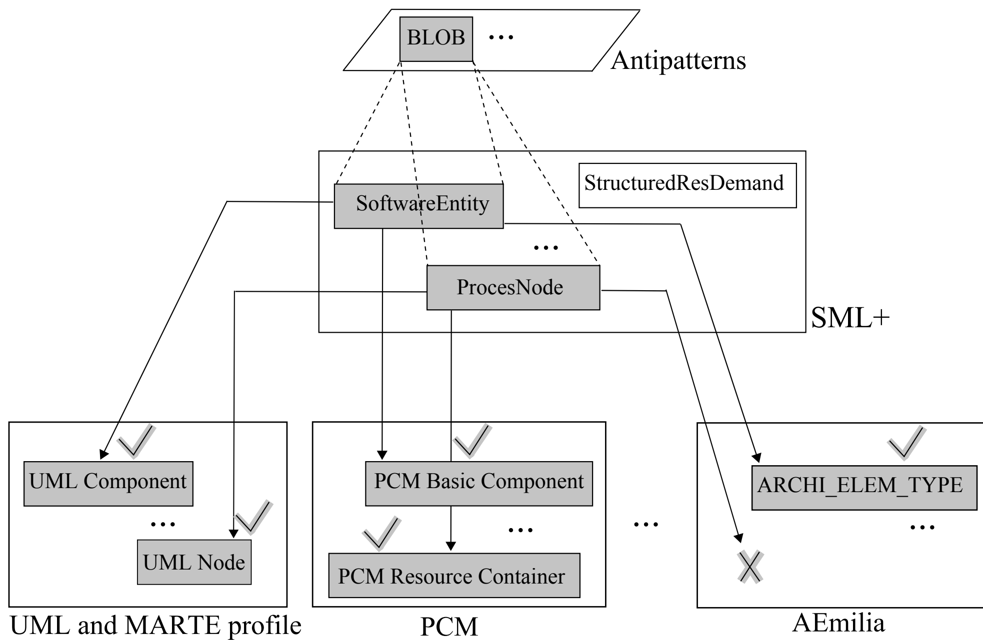


Figure 4.18: Translating antipatterns into concrete modeling languages.

when a restricted set of model elements is translated). The mappings between SML+ and the concrete modeling languages we consider in this thesis (i.e. UML and MARTE profile, PCM) are fully described in Chapter 5.

In the following we briefly present the model-driven advanced techniques that support the antipatterns-based process.

Weaving models [29] can be defined by mapping the concepts of SML+ into the corresponding concepts of a concrete modeling language (as done in [94] for different purposes, though). Weaving models represent a useful instrument in software modeling, as they can be used for setting fine-grained relationships between models or metamodels and for executing operations on them based on the link semantics.

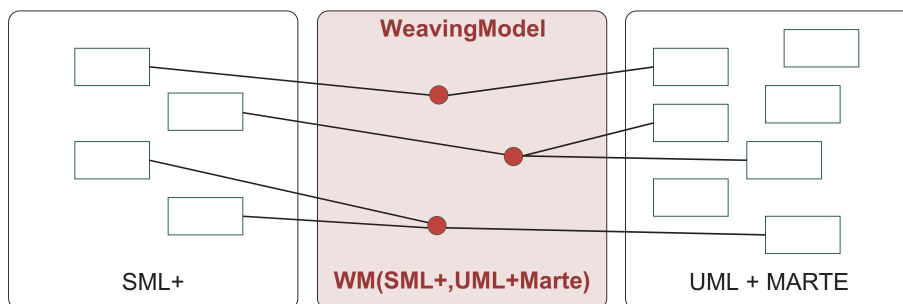


Figure 4.19: Metamodel instantiation via weaving models.

Figure 4.19 depicts how weaving models define the correspondences among two metamodels, hence the concepts in SML+ will be mapped on those in the concrete notation (e.g. UML and MARTE profile).

The benefit of using weaving models is that they can be used in automated transformations to generate other artifacts. In fact it is possible to define high-order transformations (HOT) that, starting from the weaving models, are able to generate metamodel-specific transformations embedding the antipattern in the actual concrete modeling language.

Figure 4.20 shows how to automatically generate antipatterns models in concrete modeling languages with the usage of weaving models. The metamodel we propose for antipatterns is PAML containing SML+ as (neutral) enriched software modeling language (see box $PAML_{SML+}$ of Figure 4.20). We recall that PAML is constituted by two main parts, i.e. the antipattern specification and the model elements specification grouped in SML+ (see Section 4.1). Performance antipatterns are defined as models (see Section 4.2) conform to (i.e. the arrow *c2*) the PAML metamodel (see box AP_{SML+} of Figure 4.20). Antipatterns models in concrete modeling languages (see box $AP_{UML+MARTE}$ of Figure 4.20) can be automatically generated by using the high-order transformation T that takes as input the weaving model WM specifying correspondences between *SML+* and the concrete notation under analysis (e.g. *UML+Marte*) metamodels.

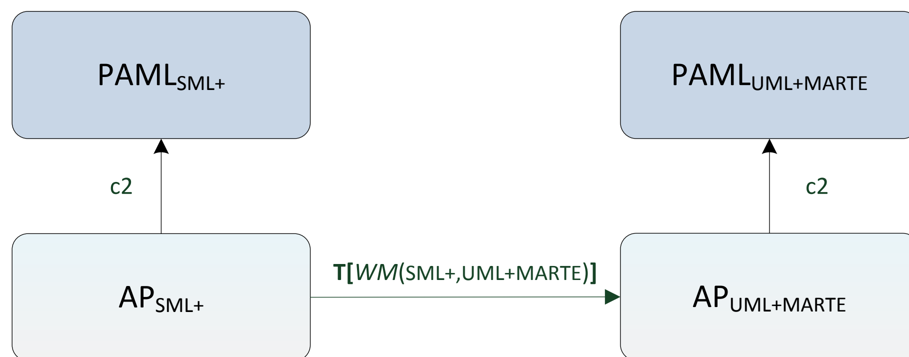


Figure 4.20: Weaving model over different software modeling languages.

A further experimentation has been conducted in UML and Marte profile where antipatterns can be naturally expressed by means of OCL [107] expressions, i.e. model queries with diagrammatic notations that correspond to first-order predicates. In particular each antipattern model conforming to PAML can be given as OCL-based semantics, in a similar way of [126], which interrogates the elements of a software architectural model to detect antipatterns. The approach we propose is aimed at automatically generating the OCL detection code from the antipattern specification (see Figure 4.21).

The leftmost part of the Figure 4.21 reports again the PAML metamodel (see box $PAML_{MM}$) and performance antipatterns models (see box $PAML_M$). First, antipatterns models are translated into intermediate models conforming to the OCL metamodel (see box OCL_{MM}) with a model-to-model transformation, i.e. *PAML2OCL* in Figure 4.21. The OCL code is generated by using a model-to-text transformation, i.e. *OCLextractor* in Figure 4.21. OCL code is finally used to *interrogate* software architectural model elements, thus to actually perform the antipattern detection. Note that the PAML metamodel provides semantics in terms of OCL: a semantic anchoring [37] is realized by means of automated transformations that map each antipattern model to an OCL expression.

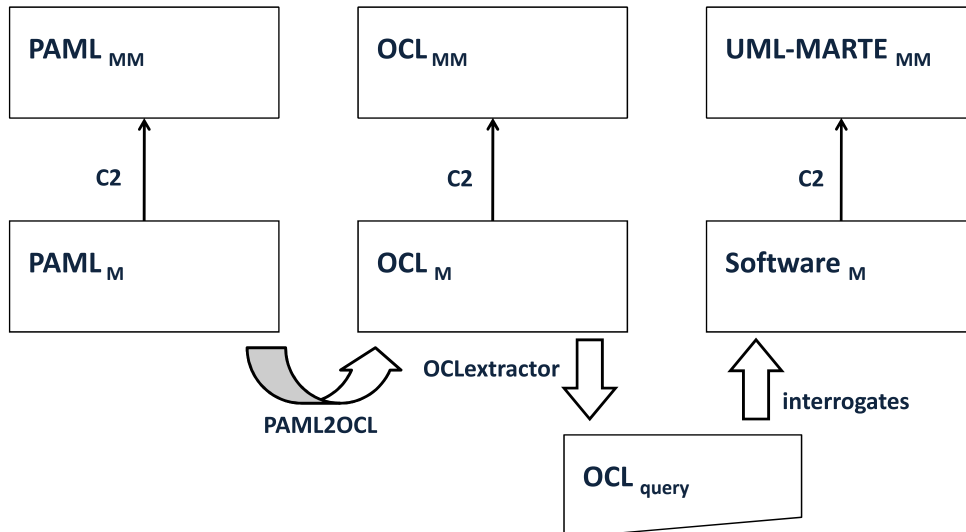


Figure 4.21: Tranforming PAML-based models in OCL queries.

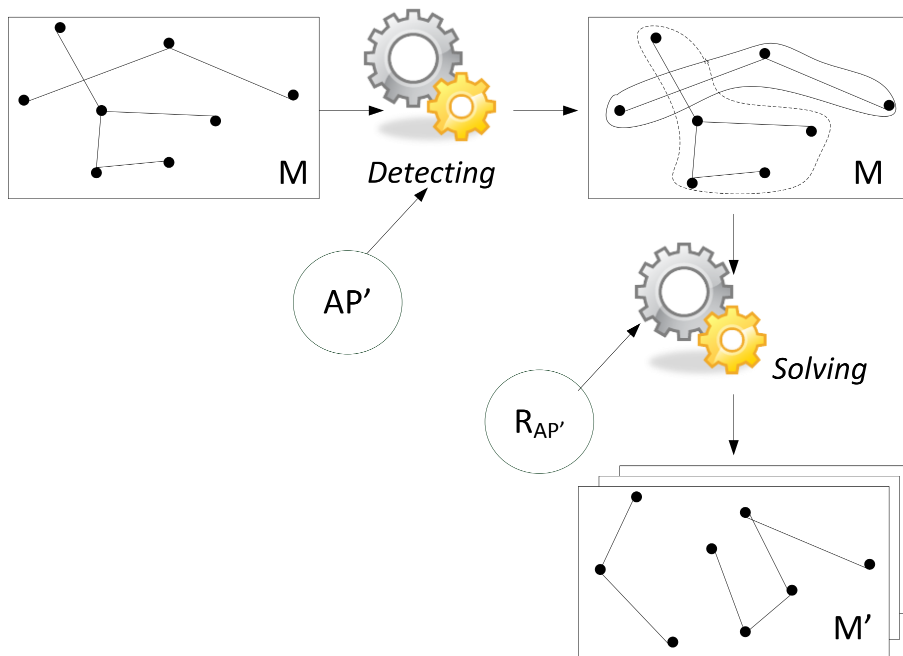


Figure 4.22: Feedback Generation approach by means of model-driven techniques.

A vision of the overall model-based framework is depicted in Figure 4.22.

A performance antipattern model AP is firstly translated into its concrete modeling language (e.g. UML and MARTE profile) counterpart AP' by means of a high order transformation, as shown in Figure 4.20. This latter representation is able to filter a software architectural model M and detect occurrences of the antipatterns.

As already said, an antipattern consists of a pattern (whose occurrence in a software architectural model is considered a bad practice) and a refactoring that possibly solves the performance problem. Therefore, if an antipattern AP has been detected in an architectural model M, then a the refactoring R_{AP} must be applied. Similarly to the antipattern model, high order transformations can be used to translate the refactoring in the concrete modeling languages to obtain $\mathrm{R}_{AP'}$. The application of the refactoring to the model M generates a new software architectural model M' where possibly the performance problems has been removed or neutralized.

The problem of refactoring architectural models is intrinsically complex and requires specialized algorithms and notations to match the abstraction level of models [93]. Recently, in [39, 113] two similar techniques have been introduced to represent refactorings as difference models. Interestingly these proposals combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches.

The specification of the antipatterns solutions can be defined as a set of refactoring actions that can be applied on the architectural model. To this end, *difference models* [39, 113] can be used to represent modifications, thus to support the activity of solving antipatterns.

CHAPTER 5

DETECTING AND SOLVING ANTIPATTERNS IN CONCRETE MODELING LANGUAGES

The goal of this Chapter is to examine performance antipatterns within concrete modeling languages, i.e. the Unified Modelling Language (UML) [12] and Marte profile [13], and the Palladio Component Model (PCM) [22]. Starting from the Performance Antipatterns Modeling Language (PAML) we define a set of mappings with UML and PCM metamodel elements in order to automate the definition of *rules* detecting antipatterns in UML and PCM models respectively. Since PAML currently deals only with the antipattern problem specification, the solution of antipatterns in concrete modeling languages has been performed by defining a set of refactoring *actions* expressed in terms of UML and PCM metamodel elements.

5.1 UML AND MARTE PROFILE

The goal of this Section is to examine performance antipatterns within the Unified Modelling Language (UML) [12], which is a language with a very broad scope that covers a large and diverse set of application domains. The UML profile for MARTE [13] (Modeling and Analysis of Real-Time and Embedded systems) provides support for specification, design, and verification/validation stages.

5.1.1 FOUNDATIONS

UML [12] groups metamodel elements into language units. A language unit consists of a collection of tightly coupled modeling concepts that provide users with the power to represent aspects of the system under study according to a particular paradigm or formalism. The stratification of language units is used as the foundation for defining compliance in UML. Namely, the set of modeling concepts of UML is partitioned into horizontal layers of increasing capability called compliance levels. Compliance levels cut across the various language units, although some language units are only present in the upper levels.

For ease of model interchange, there are just four compliance levels defined for the whole of UML: (i) Level 0 (L0). This compliance level is formally defined in the UML Infrastructure. It contains a single language unit that provides for modeling the kinds of class-based structures encountered in most popular object-oriented programming languages; (ii) Level 1 (L1). This level adds new language units and extends the capabilities provided by Level 0. Specifically, it adds language units for use cases, interactions, structures, actions, and activities; (iii) Level 2 (L2). This level extends the language units already provided in Level 1 and adds language units for deployment, state machine modeling, and profiles; (iv) Level 3 (L3). This level represents the complete UML (see Figure 5.1). It extends the language units provided by Level 2 and adds new language units for modeling information flows, templates, and model packaging.

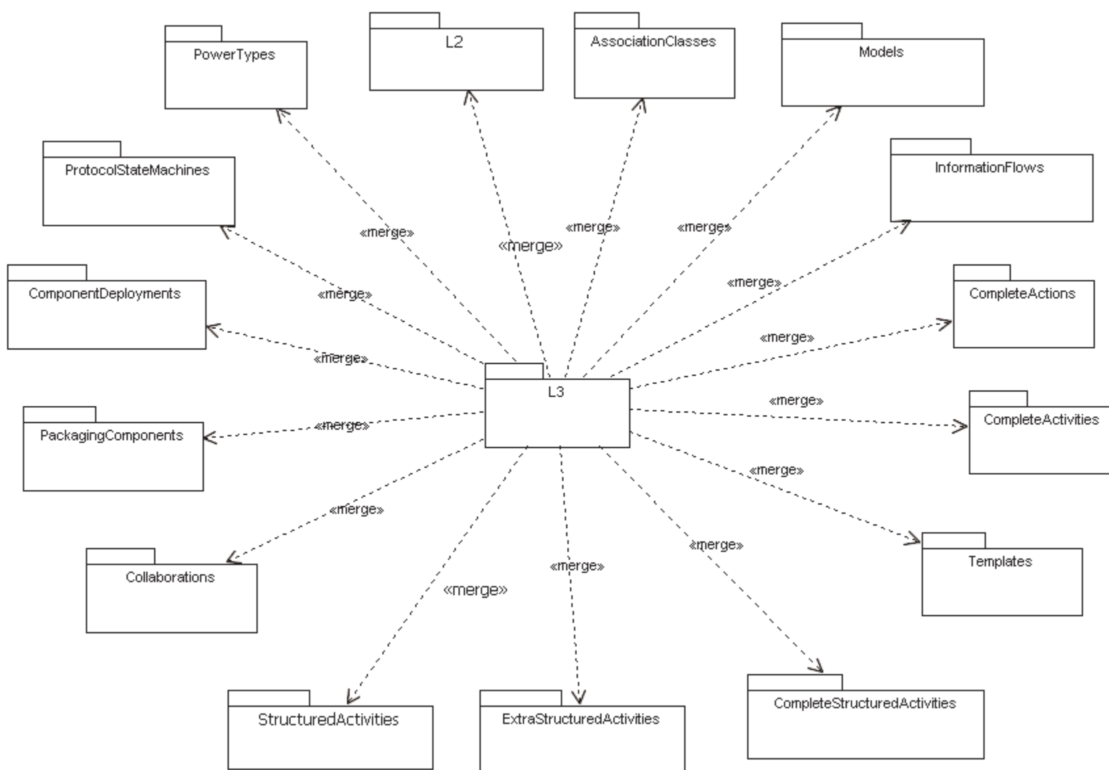


Figure 5.1: UML Compliance Level 3 top-level package merges.

MARTE [13] provides foundations for model-based description of real time and embedded systems. These core concepts are then refined for both modeling and analyzing concerns. Modeling parts provides support required from specification to detailed design of real-time and embedded characteristics of systems. MARTE supports model-based analysis. In fact, it provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance, hence it is of our interest for capturing properties of *performance antipatterns*. However, it also defines a general analysis framework intended to refine/specialize other kinds of non-functional analysis.

The MARTE profile, which replaces the current profile for Schedulability, Performance,

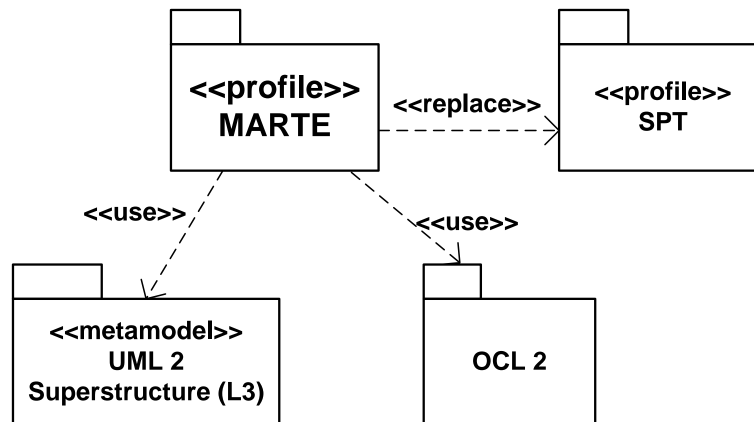


Figure 5.2: Informal description of the MARTE dependencies with other OMG standards.

and Time [14], is one of related OMG [7] specifications (see Figure 5.2). The most obvious of these is the UML 2 Superstructure specification [12], which is the basis for any UML profile. MARTE uses the OCL 2.0 specification [107] for specifying constraints.

5.1.2 DETECTING ANTIPATTERNS WITH THE MAPPING OF PAML ONTO UML

This Section is aimed at defining an explicit semantic link between the Software Modelling Language (SML+) we proposed (see Section 4.1) and UML metamodel elements. The mapping is finally used to deduce the antipattern-based *rules* used to perform the detection of antipatterns in UML models.

Tables 5.1, 5.2, 5.3 respectively report the mappings of SML+ *Static*, *Dynamic*, *Deployment* Views into UML metamodel elements: each entry specifies the meta-class, and its meta-attributes are reported in brackets. More details on the attributes are explained in the following.

SoftwareEntity.isDB- it corresponds to a UML `DataStoreNode`, since it models a central buffer node for non-transient information. It is introduced to support earlier forms of data flow modeling in which data is persistent, and we use it to model databases.

SoftwareEntity.capacity- the MARTE `paRunTInstance` stereotype provides an explicit connection between a locality or role in a behavior definition and a run time instantiation of a process. It defines properties of such process like the *poolSize* that is meant to store the number of threads for the process.

Relationship.multiplicity- it refers to the number of `Usage` or `Realization` dependencies with which a UML `Component` is connected.

Operation.probability- an operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior. The MARTE

SML+ Static View Model Element	*UML* metamodel element
SoftwareEntity (isDB, capacity)	UML Component (DataStoreNode, MARTE ≪paRunTInstance≫ {poolsize})
SoftwareEntityInstance	UML InstanceSpecification
Relationship (multiplicity)	UML Dependency (clientDependency, supplierDependency)
Operation (probability)	UML Operation (MARTE ≪paStep≫ {prob})
OperationInstance	UML Operation with MARTE ≪PaRunTInstance≫
StructuredResourceDemand	MARTE ≪GaAcqStep≫
BasicResourceDemand (type, value)	MARTE ≪GaAcqStep≫ {acqRes, resUnits}
PerformanceMetrics (throughput, responseTime)	MARTE ≪BehaviorScenario≫ {throughput, respTime}

Table 5.1: SML+ Static View Model Elements and their mapping to UML metamodel elements.

`paStep` stereotype allows to introduce performance annotations such as the probability of a branch that is stored in the *prob* attribute.

BasicResourceDemand.type, BasicResourceDemand.value- the MARTE `GaAcqStep` stereotype maps the `AcquireStep` domain element that is used to acquire a resource. In particular, two attributes are defined: (i) *acqRes* refers to the resource to be acquired within the step execution; (ii) *resUnits* refers to the number of units the resource is acquired within the step execution.

PerformanceMetrics.throughput, PerformanceMetrics.responseTime- these performance indices are evaluated for two levels of granularity: (i) `Operation` level; (ii) `Service` level. The MARTE `BehaviorScenario` stereotype defines the behavior in response to a request event, including the sequence of steps and their use of resources. The performance indices we use are: *throughput*, i.e. frequency of the operation completion, and *respTime*, i.e. end-to-end delay of part of an operation. The MARTE `GaScenario` stereotype maps the `BehaviorScenario` by capturing system-level behavior, and attaches allocations and resource usages to it. The performance indices we use are: *throughput*, i.e. the mean rate of completion of the scenario, and *respT*, i.e. the time duration from start to completion, for one scenario execution.

Service.exchangeFormats- the MARTE `GaEventTrace` stereotype maps the `EventTrace` domain element that represents the trace of events that can be the source for the request event stream, and the *format* attribute indicates the format of the event trace, i.e. how the string content can be interpreted.

Behavior.probability- similarly to the operation probability, we use the MARTE `paStep` stereotype that allows to introduce performance annotations such as the probability of a branch that is stored in the *prob* attribute.

SML+ Dynamic View Model Element	*UML* metamodel element
Service (exchangeFormats)	UML Interaction with MARTE ≪GaScenario≫ (MARTE ≪GaEventTrace≫ {format})
Behavior (probability)	UML Interaction (MARTE ≪paStep≫ {prob})
Message (multiplicity, type, maxMsgsSize, sizeMsgUnit, numRemMsgs, numRemInstances, isCreateObjectAction, isDestroyObjectAction)	UML Message (MessageOccurrenceSpecification, synchCall, MARTE ≪GaCommStep≫ {msgSize}, Deployment Diagram, Deployment Diagram, CreateObjectAction, DestroyObjectAction)

Table 5.2: SML+ Dynamic View Model Elements and their mapping to UML metamodel elements.

Message.multiplicity- it refers to the number of `Messages` between `Lifelines` in the context of an `Interaction`. A `Message` is a `NamedElement` that defines one specific kind of communication in an `Interaction`, it associates two `MessageOccurrenceSpecifications`, i.e. one sending and one receiving. Note that it refers to the function $F_{numMsgs}$ (see Table 3.2).

Message.type- the UML `Message` contains an attribute (*messageSort*) capturing the sort of communication reflected by the Message. `MessageSort` is an enumerated type that identifies the type of communication action used to generate the message. The enumeration values are: *synchCall* (the message was generated by a synchronous call to an operation); *asynchCall* (the message was generated by an asynchronous call to an operation); *asynchSignal* (the message was generated by an asynchronous send designating the creation of another lifeline object); *deleteMessage* (the message designating the termination of another lifeline); *reply* (it is a reply message to an operation call).

Message.maxMsgsSize- the MARTE `GaCommStep` stereotype maps the `CommunicationStep` domain element, and it is an operation that conveys exchange of messages. The *msgSize* attribute is meant to store the size of the message. Note that it refers to the function $F_{maxMsgSize}$ (see Table 3.2).

Message.numRemMsgs, Message.numRemInstances- both these data can be obtained by looking at the UML Deployment Diagram, since it maps the UML `Components` that manifest as `Artifacts` and deployed on UML `Nodes`, thus to give information about which components are remotely deployed. Note that they refers to the functions $F_{numRemMsgs}$ and $F_{numRemInst}$ (see Table 3.2).

Message.isCreateObjectAction, Message.isDestroyObjectAction- a communication can be, for example, raising a signal, invoking an operation, creating or destroying an instance. These two latter communications can be performed by using `CreateObjectAction`, i.e. an action that creates an object, and `DestroyObjectAction`, i.e. an action that destroys an object.

HardwareEntity.type- the UML `Node` is a computational resource upon which artifacts

SML+ Deployment View Model Element	*UML* metamodel element
HardwareEntity (type, maxDevicesUtil)	UML Node (MARTE ≪Hw_Processor≫ / MARTE ≪Hw_Memory≫, MARTE ≪GaExecHost≫ {utilization})
ProcesNode (serviceTime, waitingTime, maxDevicesUtil, maxQueueLength, maxNetworkLinksUtil, minNetworkLinksCapacity)	UML Node (MARTE ≪ResourceUsage≫ {execTime}, MARTE ≪RtUnit≫ {poolWaitingTime}, MARTE ≪GaExecHost≫ {utilization}, MARTE ≪RtBehavior≫ {queueSize}, MARTE ≪GaCommHost≫ {utilization}, MARTE ≪GaCommHost≫ {throughput})
HardwareDevice (utilization, queueLength)	UML Node (MARTE ≪GaExecHost≫ {utilization}, MARTE ≪RtBehavior≫ {queueSize})
NetworkLink (usedBandwidth, capacity, bitRate)	UML Node (MARTE ≪GaCommHost≫ {utilization}, MARTE ≪GaCommHost≫ {throughput}, -)
Params (dbConnections, webConnections, pooledResources, concurrentStreams)	MARTE ≪paRunTInstance≫ {poolsize = (param=value)})

Table 5.3: SML+ Deployment View Model Elements and their mapping to UML metamodel elements.

can be deployed for execution. In the UML metamodel, a `Node` is a subclass of `Class`; it is associated with a `Deployment` of an `Artifact` and with a set of `Elements` that are deployed on it. These `PackageableElements` are involved in a `Manifestation` of an `Artifact` that is deployed on the `Node`. There may be `Nodes` that are *nested* within the `Node`. The MARTE `HW_Processor` stereotype is meant to indicate a computing resource, it belongs to the HW_Computing package that typically implements some instruction sets and adopts branch prediction policies. The MARTE `HW_Memory` stereotype is meant to indicate any form of data storage during some interval of time, it belongs to the HW_Storage package that typically implements owns caches and corresponding memory management units.

HardwareEntity.maxDevicesUtil- the utilization of hardware devices is represented by means of the MARTE stereotype `GaExecHost` addressed to UML `Nodes`, i.e. its *utilization* attribute. Note that it refers to the function $F_{maxHwUtil}$ (see Table 3.2).

ProcesNode.serviceTime- the MARTE `ResourceUsage` stereotype maps both `ResourceUsage` and `UsageTypedAmount` domain elements, it links resources with concrete demands of usage over them; the *execTime* attribute stores the time that the resource is in use due to its usage.

ProcesNode.waitingTime- the MARTE `RtUnit` stereotype owns at least one schedulable resource. The real-time unit may either wait indefinitely for a resource to be released, or wait only a given amount of time (specified by its *poolWaitingTime* attribute).

ProcesNode.maxQueueLength- the queue length of a UML `Node` is represented by means of the MARTE stereotype `RtBehavior`, i.e. its *queueSize* attribute. Note that it refers to the function F_{maxQL} (see Table 3.2).

NetworkLink.usedBandwidth, NetworkLink.bitRate- the UML `Nodes` can be interconnected through communication paths to define network structures. The MARTE `GaCommHost` stereotype maps the `CommunicationHost` domain element, it is used for denoting a physical communication link. Two attributes are defined to represent its properties: the *throughput* and the *utilization* denoting, respectively, the actual throughput and utilization of the host they refer to.

Params- they are not directly available in UML, but they can be defined by associating the MARTE `paRunTInstance` stereotype to a process, and as a special type of *poolSize* elements that influence the run time instantiation of such process.

In the following we present an example that demonstrates how to use the mapping of PAML onto UML and Marte profile to deduce the antipattern-based *rules*.

Concurrent Processing Systems Starting from the PAML-based antipattern model (see Figure 4.7) the following rules are deduced.

CPS.deployment.queue Rule- there is at least one UML `Node`, e.g. PN_1, that has a high average queue length. This rule is evaluated by extracting the `queueSize` tagged value of the MARTE stereotype `RtBehavior`, and checking if it is greater or equal to the threshold $Th_{maxQueue}$.

CPS.deployment.unbalancedCpus Rule- there is at least one UML `Node` $cpuPN_1$ nested in the UML `Node` PN_1 that has a high average utilization. This rule is evaluated by extracting the `utilization` tagged value of the MARTE stereotype `Hw_Processor`, and checking if it is greater or equal to the threshold $Th_{cpuMaxUtil}$. There is at least another UML `Node` $cpuPN_2$ nested in a UML `Node`, e.g. PN_2, that has a low average utilization. This rule is evaluated by extracting the `utilization` tagged value of the MARTE stereotype `Hw_Processor`, and checking if it is lower than the threshold $Th_{cpuMinUtil}$.

CPS.deployment.unbalancedDisks Rule- there is at least one UML `Node` $diskPN_1$ nested in the UML `Node` PN_1 that has a high average utilization. This rule is evaluated by extracting the `utilization` tagged value of the MARTE stereotype `Hw_Memory`, and checking if it is greater or equal to the threshold $Th_{diskMaxUtil}$. There is at least another UML `Node` $diskPN_2$ nested in a UML `Node`, e.g. PN_2, that has a low average utilization. This rule is evaluated by extracting the `utilization` tagged value of the MARTE stereotype `Hw_Memory`, and checking if it is lower than the threshold $Th_{diskMinUtil}$.

Each (PN_1, PN_2) $\in$ the set of all the UML `Nodes` represents a Concurrent Processing Systems antipattern in a UML model if it matches the *CPS.deployment.queue Rule* AND the *CPS.deployment.unbalancedCpus Rule* OR the *CPS.deployment.unbalancedDisks Rule*, as stated in the PAML-based antipattern model (see Figure 4.7).

5.1.3 Solving antipatterns in UML

From the informal representation of the *solution* (see Table 3.1) a set of *actions* is built, where each action addresses part of the antipattern solution specification.

Blob - the solution of this antipattern can be performed with the following actions.

DelegateWork Action- delegate the business logics from the Blob UML `Component` to the other ones by decreasing the number of `Usage` or `Realization` dependencies. In the UML Component Diagram it is possible to move one or more `Operations` from the Blob component to the surrounding ones by deleting some dependencies. Such changes must be reflected in UML Sequence Diagram(s) by moving operations in the new `Lifelines`' owners.

Redeploy Action- if the Blob `Component` (c_x) and the surrounding ones (c_{x1}, c_{x2}, $\ldots$, c_{xn}) are distributed on different UML `Nodes`, the re-deployment of components can avoid remote communications and should automatically improve the utilization of the involved nodes. Such action can be performed in the UML Deployment Diagram by changing the relation of `deployment` between UML `artifacts` and UML `Nodes`.

IncreaseSpeed Action- if the Blob `Component` (c_x) and the surrounding ones (c_{x1}, c_{x2}, $\ldots$, c_{xn}) are deployed on the same UML `Node`, the increasing speed of such node should automatically improve its utilization. Such action can be performed in the UML Deployment Diagram by modifying the value of the tagged value `speedFactor` of the Marte stereotype `ProcessingResource`. Another option is to analyze all the UML `Node` instances in order to decide if it is better to re-deploy all the involved components on another node, and then increase the speed of the latter one by a smaller percentage.

Mirror Action- the Blob `Component` (c_x) and the surrounding ones (c_{x1}, c_{x2}, $\ldots$, c_{xn}) can be mirrored as new components and deployed into another UML `Node` (an existing one, if possible), thus to balance the workload of requests incoming to the system.

Concurrent Processing Systems - the solution of this antipattern can be performed with the following actions.

BalanceLoad Action- if the UML `Nodes` n_x and n_y offer the same `Interfaces`, change the scheduling algorithms and distribute in a balanced way (from n_x to n_y) the requests for such services by modifying the probability to be called.

Mirror Action- mirror the UML `Components` of the `Node` n_x into n_y and balance the workload, so that the requests incoming to the system are distributed to both `Nodes`.

MostCritical Action- identify the `Component` of the `Node` n_x that has the highest resource demand of the critical type t, and re-deploy it in the `Node` n_y.

Redeploy Action- re-deploy some `Components` from the `Node` n_x to n_y. Such action

can be performed by taking into account a set of system properties or their combination, as argued in the following.

The first option is aimed at re-deploying the components on the basis of their resource demand types. The re-deployment of components can be performed by evaluating the node n_y and deciding whenever it is better to re-deploy CPU-critical (high computation demand, i.e. the MARTE `hostDemand` attribute). components and/or disk-critical (high storage demand, i.e. the MARTE `allocatedMemory` attribute) ones.

The second option is aimed at re-deploying components on the basis of the utilization of nodes under analysis. The nodes can be considered more or less critical if: a) there is at least one nested `Node` of type $t*$, i.e. CPU (with the MARTE `Hw_Processor` stereotype), disk (with the MARTE `Hw_Memory` stereotype), that exceeds a threshold value; b) all the nested `Nodes` of type $t*$ exceed a threshold value.

The third option is aimed at re-deploying components on the basis of their communication. Network links can considered more or less critical if: a) two components communicate through a node whose `Communication Path` exceeds a threshold value, but such components also use other nodes that do not provide any violation; b) all the nodes provide a violation.

Pipe and Filter Architectures - the solution of this antipattern can be performed with the following actions.

IncreaseCapacity Action- increase the pool size of the `Component` providing the slowest filter, thus requests with a lower resource demand are not delayed.

SplitAndRedeploy Action- split the slowest filter into two operations and re-deploy one of them thus to facilitate the processing of incoming requests to the system, especially the ones with a lower resource demand.

Extensive Processing - the solution of this antipattern can be performed with the following actions.

IncreaseCapacity Action- increase the capacity of `Components` locking incoming requests, thus to increase system concurrency.

UnblockExecution Action- change the scheduling algorithm of the resource and/or re-deploy one of the `Components` containing op_a or op_b thus they do not queue for the same `Node` anymore.

Circuitous Treasure Hunt - the solution of this antipattern can be performed by refactoring the database component (i.e. UML `DataStoreNode`) in its internal structure. We leave this task to the designer that better knows how to organize the structure of the database in such a way that incoming requests avoid to access too many tables.

Empty Semi Trucks - the solution can be performed with the following action.

Batching Action- optimize the usage of the bandwidth by reducing the number of sent messages through the batching, i.e. joining all small messages in few messages of a bigger size, thus to reduce the MARTE `commTxOvh` and `commRcvOvh` attributes value, i.e. the host demand for sending and receiving messages respectively.

Tower of Babel - the solution of this antipattern can be performed by reducing the number of exchanging formats, thus to optimize the overhead they require.We leave the task of restructuring such situations to the designer.

One-Lane Bridge - the solution can be performed with the following action.

IncreaseCapacity Action- increase the capacity of the `Nodes`. Such action can be performed in the UML Deployment Diagram by modifying the value of the tagged value `speedFactor` of the Marte stereotype `ProcessingResource`. A smarter methodology can be devised in order to optimize the capacity by evaluating the minimal multiplicity able to solve performance issues.

Excessive Dynamic Allocation - the solution of this antipattern can be performed by avoiding an unnecessary creation and destruction objects. We leave the task of restructuring such situations to the designer.

Traffic Jam - the solution of this antipattern can be performed with the following action.

IncreaseCapacity Action- increase the capacity of the `Node` on which the `Interaction` causing traffic jam is executed. Such action can be performed in the UML Deployment Diagram by modifying the value of the tagged value `speedFactor` of the Marte stereotype `ProcessingResource`.

The Ramp - the solution of this antipattern can be performed by notifying the designer that a `Interaction` is increasingly getting worse (from a performance perspective), i.e. increasing response time and decreasing throughput over time. We leave the task of restructuring such situations to the designer.

Note that the *More is Less* antipattern cannot be detected in UML models, hence we exclude it from the definition of refactoring actions.

5.2 Palladio Component Model

The goal of this Section is to examine performance antipatterns within the Palladio Component Model (PCM) [22], which is a domain specific modeling language to describe component-based software architectures.

5.2.1 FOUNDATIONS

The Palladio Component Model (PCM) [22] is a domain-specific modeling language for component-based software architectures. The PCM enables the explicit definition of the i) components , ii) architecture, iii) allocation, and iv) usage of a system in respective artifacts, which together have a PCM instance.

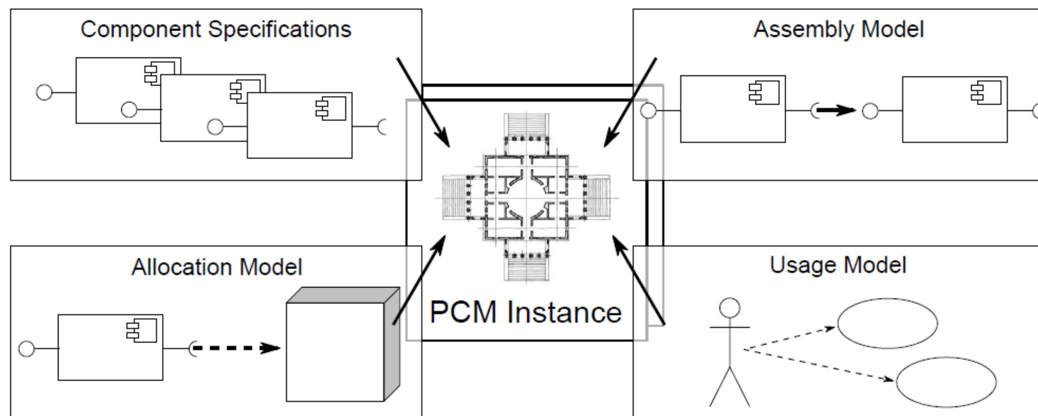


Figure 5.3: Artifacts of a PCM instance.

Figure 5.3 shows the main artifacts of a PCM instance: (i) *Component Specifications* contain an abstract, parametric description of components; furthermore, the behavior of the components is specified using a UML activity diagram similar syntax; (ii) *Assembly Model* is meant to define the software architecture; (iii) *Allocation Model* groups the resource environment and the allocation of components to hardware resources; (iv) *Usage Model* specifies usage scenarios: to each user, at least one scenario applies by defining the frequency and the sequence of interactions with the system, i.e. the system functionalities used with an entry level system call.

In order to quickly convey the concepts of the PCM, a simple example in Figure 5.4 contains some of the PCM model elements that are important for antipattern detection and solution. In what follows, the model elements are marked with `typewriter font`. Note that only features relevant to the antipatterns are shown here, other PCM features can be found in [22].

A software system in the PCM is modeled as a set of components (`Basic Components` C1, C2 in Figure 5.4). Components offer `Interfaces`. In the example, `Basic Component` C1 offers `Interface` I1, while `Basic Component` C2 offers `Interface` I2. Additionally, components can require interfaces. In the example, C1 requires the `Interface` I2. Components are assembled as a `System` by connecting provided and required `Interfaces`. For example, the `Interface` I2 provided by component C2 satisfies C1's requirement of that `Interface`.

A PCM model also contains the mapping of software components to hardware, called `Allocation`. Hardware platforms are modeled as `Resource Containers`, which

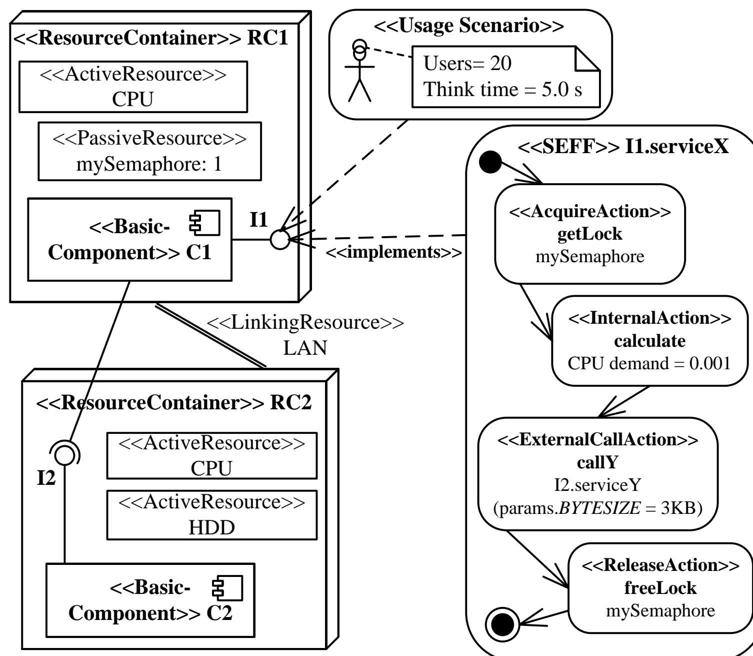


Figure 5.4: Example of a PCM model.

can contain `Active Resources`, such as CPU and hard disk (HDD), or `Passive Resources`, such as semaphores or thread pools. In Figure 5.4, a semaphore `Passive Resource` with capacity 1 is modeled in `Resource Container` RC1. `Active Resources` have additional properties not shown here, such as a processing rate (how many demand units per second they process) and scheduling policies (such as FCFS or processor sharing). The mapping of components to `Resource Containers` is visualised by placing the components inside the container. `Resource Containers` are connected by `Linking Resources`, whose timing behavior is determined by the size of sent data.

`Service Effect Specifications` (`SEFFs`) describe the behavior of the services offered by the `Basic Components`. A `SEFF` contains a sequence of actions. `External Call Actions` model calls to required interfaces. For example, serviceX of component C1 calls the serviceY of interface I2. As C1 is connected to C2 with this interface, the call is directed to C2's serviceY. Optionally, the size of the passed data can be specified with a `BYTESIZE Characterisation`, which is used to determine the linking resource load. `Internal Actions` specify a resource demand to an `Active Resource`, such as a CPU or a hard disk (HDD). In the example, serviceX of component C1 has a CPU demand of 0.01 each time it is called. `Acquire Actions` and `Release Actions` model the use of `Passive Resources` in the PCM. Control flow structures are modeled with `LoopActions`, `BranchActions`, and `ForkActions` (not shown in Figure 5.4).

5.2.2 Detecting antipatterns with the mapping of PAML onto PCM

This Section is aimed at defining an explicit semantic link between the Software Modelling Language (SML+) we proposed (see Section 4.1) and PCM metamodel elements. The mapping is finally used to deduce the antipattern-based *rules* used to perform the detection of antipatterns in PCM models.

Tables 5.4, 5.5, 5.6 respectively report the mapping of SML+ *Static*, *Dynamic*, *Deployment* Views into PCM metamodel elements: each entry specifies the meta-class and its meta-attributes are reported in brackets. More details on the attributes are explained in the following.

SML+ Static View Model Element	*PCM* metamodel element
SoftwareEntity (isDB, capacity)	PCM Basic Component (-, Passive Resource)
SoftwareEntityInstance	PCM Allocation Component
Relationship (multiplicity)	PCM Assembly Connector (Required Role/ Provided Role)
Operation (probability)	PCM Service Effect Specification (Usage Model), associated to a PCM Basic Component
OperationInstance	PCM Service Effect Specification, associated to a PCM Allocation Component
StructuredResourceDemand	PCM Resource Demanding SEFF
BasicResourceDemand (type, value)	PCM Parametric Resource Demand (Processing Resource Type, Processing Resource Specification)
PerformanceMetrics (throughput, responseTime)	PCM SimuCom

Table 5.4: SML+ Static View Model Elements and their mapping to PCM metamodel elements.

SoftwareEntity.isDB- it is not directly available in the PCM, but it could be added by introducing a *decorator model* [61] that explicitly marks the database `Basic Components`.

SoftwareEntity.capacity- the capacity of a PCM `Basic Component` can be modeled with a `Passive Resource`: if each of the `SEFF`s of the basic component starts with an `Aquire Action` and ends with a `Release Action` to that passive resource, then the capacity of such resource corresponds to the capacity of the basic component.

Relationship.multiplicity- it maps to the number of `Required Roles` of a PCM Basic Component in case such component is involved in the relationship as client, whereas it maps to the number of `Provided Roles` of a PCM Basic Component in case such component is involved in the relationship as supplier.

Operation.probability- the operation probability is not known at the static structure of a PCM instance, however it can be derived as soon as the PCM `Usage Model` is known

by analysing the control flow with the PCM dependency solver.

BasicResourceDemand.type, BasicResourceDemand.value- it refers to the amount of processing requested to a certain type of resource in a parameterized way, i.e. `Processing Resource Type` such as CPU or hard disk, and `Processing Resource Specification` such as 5 Ghz or 20 MByte/s.

PerformanceMetrics.throughput, PerformanceMetrics.responseTime- the transformation from the PCM software architectural model to the performance model [90] generates the simulation code for the PCM SimuCom [22], i.e. a simulation tool by which the performance model is simulated to obtain the performance indices of interest such as throughput, response time, utilization, etc.

SML+ Dynamic View Model Element	*PCM* metamodel element
Service (exchangeFormats)	PCM Usage Scenario (-)
Behavior (probability)	-
Message (multiplicity, type, maxMsgsSize, sizeMsgUnit, numRemMsgs, numRemInstances, isCreateObjectAction, isDestroyObjectAction)	PCM External Call Action (noOfCalls, synchronous, BYTESIZE Characterisation, Allocation Model, Allocation Model, -, -)

Table 5.5: SML+ Dynamic View Model Elements and their mapping to PCM metamodel elements.

Service.exchangeFormats- it is not directly available in the PCM, since data flow is more abstract and it does not include information on data formats. It is for this reason that the Tower of Babel antipattern cannot be currently detected in the PCM modeling language.

Behavior.probability- it is not explicitly available in the PCM, as the PCM only models the behavior of single components, so that the behavior of the system can be derived from analyses. For each part of the system, a global behavior model could be derived starting from the entry level system calls, because then a full trace through the system can be derived. The probability attribute, similarly to the operation probability, can be derived as soon as the PCM `Usage Model` is known by analysing the control flow with the PCM dependency solver.

Message.multiplicity- it can be derived for a `Basic Component` by counting the number of `External Call Actions` in the `SEFFs` it is involved.

Message.type- the PCM `External Call Actions` always model synchronous calls, i.e. the caller waits until the called service finishes, before continuing the execution itself.

Message.maxMsgsSize- the message size in PCM is specified by means of the `BYTESIZE Characterisation`, and the size of the file is expressed in bytes.

Message.numRemMsgs, Message.numRemInstances- both these data can be obtained by looking at the PCM `Allocation Model`, since it maps the PCM `Assembly Context` with `Resource Containers`, thus to give information if basic components

are remotely deployed.

Message.isCreateObjectAction, Message.isDestroyObjectAction- it is not directly available in the PCM, since no object-oriented detail is available in the current abstraction level. It is for this reason that the Excessive Dynamic Allocation antipattern cannot be currently detected in the PCM modeling language.

SML+ Deployment View Model Element	*PCM* metamodel element
HardwareEntity (type, maxDevicesUtil)	PCM Active Resource (type, PCM SimuCom)
ProcesNode (serviceTime, waitingTime, maxDevicesUtil, maxQueueLength, maxNetworkLinksUtil, minNetworkLinksCapacity)	PCM Resource Container (PCM SimuCom, PCM SimuCom, PCM SimuCom, PCM SimuCom, PCM SimuCom, PCM SimuCom)
HardwareDevice (utilization, queueLength)	PCM Active Resource (PCM SimuCom, PCM SimuCom)
NetworkLink (usedBandwidth, capacity, bitRate)	PCM Linking Resource (PCM SimuCom, PCM SimuCom, Variable Characterisation)
Params (dbConnections, webConnections, pooledResources, concurrentStreams)	-

Table 5.6: SML+ Deployment View Model Elements and their mapping to PCM metamodel elements.

Many attributes belonging to the Deployment View are derived from the PCM SimuCom [22], i.e. the simulation tool by which the performance model is simulated to obtain the performance indices of interest. In particular, we refer to *HardwareEntity.maxDevicesUtil*, *ProcesNode.serviceTime*, *HardwareDevice.utilization*, *NetworkLink.usedBandwidth*, etc.

HardwareEntity.type- in the PCM hardware platforms are modeled as Resource Containers, which can contain `Active Resources`, such as CPU and hard disk (HDD), or Passive Resources, such as semaphores or thread pools.

NetworkLink.bitRate- the PCM `Variable Characterisation` is meant to store performance critical information on the `Random Variable` attribute referring to the PCM `Linking Resource`.

Params- it is not directly available in the PCM, since data flow is more abstract and it does not include information on run time parameters such as database connections, web connections, etc. It is for this reason that the More is Less antipattern cannot be currently detected in the PCM modeling language.

Note that multiple-values antipatterns might be supported by introducing in the PCM the concept of *state* as suggested in [84], and it might be possible to inform the designer that a resource demand increasingly grows due to state changes.

In the following we present an example that demonstrates how to use the mapping of PAML onto PCM to deduce the antipattern-based *rules*.

Concurrent Processing Systems Starting from the PAML-based antipattern model (see Figure 4.7) the following rules are deduced.

Rule cps.deployment.queue - there is at least one PCM `Resource Container`, e.g. PN_1, that has a high average queue length. This rule is evaluated by extracting the queue length of its PCM `Active Resources` from the PCM simulation results, and checking if it is greater or equal to the threshold $Th_{maxQueue}$.

Rule cps.deployment.unbalancedCpus - there is at least one PCM `Active Resource` $cpuPN_1$ (with type equal to `CPU`) nested in the PCM `Resource Container` PN_1 that has a high average utilization. This rule is evaluated by extracting the utilization from the PCM simulation results, and checking if it is greater or equal to the threshold $Th_{cpuMaxUtil}$. There is at least another PCM `Active Resource` $cpuPN_2$ (with type equal to `CPU`) nested in a PCM `Resource Container`, e.g. PN_2, that has a low average utilization. This rule is evaluated by extracting the utilization from the PCM simulation results, and checking if it is lower than the threshold $Th_{cpuMinUtil}$.

Rule cps.deployment.unbalancedDisks - there is at least one PCM `Active Resource` $diskPN_1$ (with type equal to `HDD`) nested in the PCM `Resource Container` PN_1 that has a high average utilization. This rule is evaluated by extracting the utilization from the PCM simulation results, and checking if it is greater or equal to the threshold $Th_{diskMaxUtil}$. There is at least another PCM `Active Resource` $diskPN_2$ (with type equal to `HDD`) nested in a PCM `Resource Container`, e.g. PN_2, that has a low average utilization. This rule is evaluated by extracting the utilization from the PCM simulation results, and checking if it is lower than the threshold $Th_{diskMinUtil}$.

Each $(PN_1,\ PN_2) \in$ the set of all the PCM `Resource Containers` represents a Concurrent Processing Systems antipattern in a PCM model if it matches the *CPS.deployment.queue Rule* AND the *CPS.deployment.unbalancedCpus Rule* OR the *CPS.deployment.unbalancedDisks Rule*, as stated in the PAML-based antipattern model (see Figure 4.7).

5.2.3 Solving antipatterns in PCM

From the informal representation of the *solution* (see Table 3.1) a set of *actions* is built, where each action addresses part of the antipattern solution specification.

Blob - the solution of this antipattern can be performed with the following actions.

DelegateWork Action- delegate the business logics from the Blob PCM `Basic Component` to the other ones by decreasing the number of `RequiredRole` or `ProvidedRole` connectors. In the PCM Assembly Model it is possible to move one or more `SEFF`s from the Blob component to the surrounding ones by deleting some dependencies. Such changes must be reflected in PCM `SEFF`(s) by moving `External Call Actions`.

Redeploy Action- if the Blob `Basic Component` (bc_x) and the surrounding ones (bc_{x1}, bc_{x2}, $\ldots$, bc_{xn}) are distributed on different PCM `Resource Containers`, the re-deployment of components can avoid remote communications and should automatically improve the utilization of the involved resource containers. Such action can be performed in the PCM Allocation Model by changing the relation of `deployed-on` between PCM `Basic Components` and PCM `Resource Containers`.

IncreaseSpeed Action- if the Blob `Basic Component` (bc_x) and the surrounding ones (bc_{x1}, bc_{x2}, $\ldots$, bc_{xn}) are deployed on the same PCM `Resource Container`, the increasing speed of such node should automatically improve its utilization. Such action can be performed in the PCM Allocation Model by modifying the value of the `processingRate` in the `ProcessingResourceSpecification`. Another option is to analyze all the PCM `Resource Container` instances in order to decide if it is better to re-deploy all the involved components in another container, and then increase the rate of the latter one by a smaller percentage.

Mirror Action- the Blob `Basic Component` (bc_x) and the surrounding ones (bc_{x1}, bc_{x2}, $\ldots$, bc_{xn}) can be mirrored as new components and deployed into another PCM `Resource Container` (an existing one, if possible), thus to balance the workload of requests incoming to the system.

Concurrent Processing Systems - the solution of this antipattern can be performed with the following actions.

BalanceLoad Action- if the `Resource Containers` rc_x and rc_y offer the same `Interfaces`, change the scheduling algorithms and distribute in a balanced way (from rc_x to rc_y) the requests for such services by modifying the probability to be called.

Mirror Action- mirror the `Basic Components` of the `Resource Container` rc_x into rc_y and balance the workload, so that the requests incoming to the system are distributed to both `Resource Containers`.

MostCritical Action- identify the `Basic Component` of the `Resource Container` rc_x that has the highest resource demand of the critical type t, and re-deploy it in the `Resource Container` rc_y.

Redeploy Action- re-deploy some `Basic Components` from the `Resource Container` rc_x to rc_y. Such action can be performed by taking into account a set of system properties or their combination, as argued in the following.

The first option is aimed at re-deploying components on the basis of their resource demand types. The re-deployment of components can be performed by evaluating the resource container rc_y and deciding whenever it is better to re-deploy CPU-critical (high computation demand) components and/or HardDisk-critical (high storage demand) ones.

The second option is aimed at re-deploying components on the basis of the utilization of

nodes under analysis. The nodes can be considered more or less critical if: a) there is at least one nested `Active Resource` of type $t*$ (i.e. CPU, HardDisk) that exceeds a threshold value; b) all the nested `Active Resources` of type $t*$ exceed a threshold value.

The third option is aimed at re-deploying components on the basis of their communication. Linking resources can considered more or less critical if: a) two components communicate through a `Linking Resource` that exceeds a threshold value, but such components also use other linking resources that do not provide any violation; b) all the linking resources provide a violation.

Pipe and Filter Architectures - the solution of this antipattern can be performed with the following actions.

IncreaseCapacity Action- increase the capacity of the `Passive Resources` providing the acquire and release actions for the slowest filter, thus requests with a lower resource demand are not delayed.

SplitAndRedeploy Action- split the slowest filter, i.e. the PCM *protected region*, into two `SEFFs` and re-deploy one of them thus to facilitate the processing of incoming requests to the system, especially the ones with a lower resource demand.

Extensive Processing - the solution of this antipattern can be performed with the following actions.

IncreaseCapacity Action- increase the capacity of `Passive Resource` locking the `Branch Action`.

UnblockExecution Action- change the scheduling algorithm of the resource and/or re-deploy one of the `Basic Components` containing $seff_a$ or $seff_b$ thus they do not queue for the same `Active Resource` anymore.

Circuitous Treasure Hunt - the solution of this antipattern can be performed by refactoring the database basic component (i.e. annotated with a custom mark model) in its internal structure. We leave this task to the designer that better knows how to organize the structure of the database in such a way that incoming requests avoid to access too many tables.

Empty Semi Trucks - the solution can be performed with the following action.

Batching Action- optimize the usage of the bandwidth by reducing the number of sent messages through the batching, i.e. joining all small messages in few messages of a bigger size through changes to the PCM `BYTESIZE Characterisation` attribute.

One-Lane Bridge - the solution can be performed with the following action.

IncreaseCapacity Action- increase the capacity of the Passive Resources. A smarter methodology can be devised in order to optimize the capacity by evaluating the minimal multiplicity able to solve performance issues.

Traffic Jam - the solution of this antipattern can be performed with the following action.

IncreaseCapacity Action- increase the speed of the Resource Container on which the SEFF detected as responsible of the large backlog ($seff_a$) is executed. Such action can be performed in the PCM Allocation Model by modifying the value of the processingRate attribute of the PCM ProcessingResourceSpecification.

The Ramp - the solution of this antipattern can be performed by notifying the designer that a PCM SEFF is increasingly getting worse (from a performance perspective), i.e. increasing response time and decreasing throughput over time. We leave the task of restructuring such situations to the designer.

Note that the *Tower of Babel*, *Excessive Dynamic Allocation*, and *More is Less* antipatterns cannot be detected in PCM models, hence we exclude them from the definition of refactoring actions.

5.3 Summary and Lessons Learned

Table 5.7 summarizes the expressiveness of the UML and PCM modeling languages for specifying performance antipatterns. In particular, it is organized as follows: each row represents a specific antipattern and it is characterized by the *antipattern* name, if it is automatically *detectable* and *solvable* in UML and PCM models respectively.

The entries of Table 5.7 can be of three different types with the following meaning: $\surd$ (i.e. yes), $\times$ (i.e. no), and $-$ denotes that the corresponding operation does not make sense, i.e. if an antipattern cannot be detected it is obvious that it cannot be solved.

These experiences have allowed to classify the antipatterns in three categories: (i) detectable and solvable; (ii) semi-solvable (i.e. the antipattern solution is only achieved with refactoring actions to be manually performed); (iii) neither detectable nor solvable.

Table 5.7 points out that the most interesting performance antipatterns in UML and PCM are: *Blob*, *Concurrent Processing Systems*, *Pipe and Filter Architectures*, *Extensive Processing*, *Empty Semi Trucks*, *One-Lane Bridge*, and *Traffic Jam*, since they are either detectable and solvable.

Table 5.7 shows that there are currently two *semi-solvable* performance antipatterns, i.e. *Circuitous Treasure Hunt*, and *The Ramp*, since they can be detected, but they cannot be automatically solved in both UML and PCM modeling languages.

<table>
<tr><th colspan="3" rowspan="2">Antipattern</th><th colspan="2">UML</th><th colspan="2">PCM</th></tr>
<tr><th>Detectable</th><th>Solvable</th><th>Detectable</th><th>Solvable</th></tr>
<tr><td rowspan="10">Single-value</td><td colspan="2">Blob (or god class/component)</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td rowspan="3">Unbalanced Processing</td><td>Concurrent Processing Systems</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td>Pipe and Filter Architectures</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td>Extensive Processing</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td colspan="2">Circuitous Treasure Hunt</td><td>√</td><td>×</td><td>√</td><td>×</td></tr>
<tr><td colspan="2">Empty Semi Trucks</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td colspan="2">Tower of Babel</td><td>√</td><td>×</td><td>×</td><td>–</td></tr>
<tr><td colspan="2">One-Lane Bridge</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td colspan="2">Excessive Dynamic Allocation</td><td>√</td><td>×</td><td>×</td><td>–</td></tr>
<tr><td colspan="2"></td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="3">Multiple-values</td><td colspan="2">Traffic Jam</td><td>√</td><td>√</td><td>√</td><td>√</td></tr>
<tr><td colspan="2">The Ramp</td><td>√</td><td>×</td><td>√</td><td>×</td></tr>
<tr><td colspan="2">More is Less</td><td>×</td><td>–</td><td>×</td><td>–</td></tr>
</table>

Table 5.7: Performance Antipatterns detectable and solvable in UML and PCM.

Table 5.7 highlights with shaded rows that two antipatterns, i.e. *Tower of Babel* and *Excessive Dynamic Allocation* are expressed in a different way in UML and PCM. In particular, we can notice that in UML these antipatterns can be detected, but they cannot be automatically solved, whereas in PCM they are neither detectable nor solvable.

Tower of Babel is an antipattern whose bad practice is on the translation of information into too many exchange formats, i.e. data is parsed and translated into an internal format, but the translation and parsing is excessive [123]. In the PCM, data flow is more abstract and does not include information on data formats in the current abstraction level. However, it might be possible to replace the current modeling language to specify the behavioural description of services, i.e. the PCM service effect specification (SEFF), by another behavioural description language that includes such detail.

Excessive Dynamic Allocation is an antipattern whose bad practice is on unnecessarily creating and destroying objects during the execution of an application [119]. In the PCM, no object-oriented detail is available, because it is not included in the current abstraction level. However, it might be possible to detect such bad practice in PCM models that are re-engineered from byte code [91], because constructor invocations are then stored as special type of resource demands at the modeling layer.

Finally, Table 5.7 indicates that there is currently one performance antipattern, i.e. *More is Less*, neither detectable nor solvable in UML and PCM.

More is Less is an antipattern whose bad practice is on the overhead spent by the system in thrashing in comparison of accomplishing the real work [121]. Currently thrashing, in particular page faults, cannot be explicitly modeled neither in UML nor in PCM.

In UML it might be added by introducing a specific *event* (e.g. “page fault”) in the *subset type* of an `EventOccurrence` belonging to the MARTE *Causality::RunTimeContext* package (see Figure 5.5).

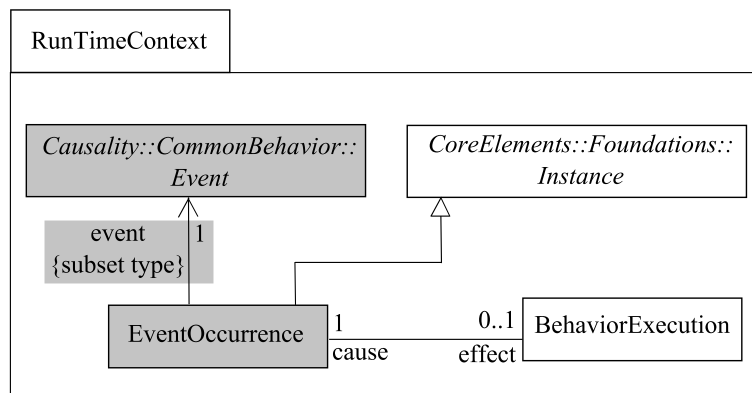


Figure 5.5: An excerpt of the MARTE *RunTimeContext* package.

In PCM it might be added by introducing layered execution environment models, as suggested in [76]. A controller infrastructure can be modeled to capture page faults as an execution environment model and it can be integrated in the PCM resource environment.

CHAPTER 6

VALIDATION

The goal of this Chapter is to demonstrate that the antipattern-based approach we propose is built on methodologies that can be applied on industrial products without delaying the software development process. Such methodologies are aimed at providing automation in the software performance process, thus to achieve an early performance validation.

In this direction we here report our experience in the analysis of two case studies whose performance issues have been solved with the usage of the antipattern solution. We discuss the advantages and the disadvantages of the applied techniques in UML and PCM, and we compare them in order to abstract towards the suitable characteristics the software systems should have.

6.1 A CASE STUDY IN UML

This Section deals with a UML example, and is organized as follows. First, Section 6.1.1 describes the UML model of the system under analysis, the so-called E-commerce System (ECS). Then, the stepwise application of our antipattern-based process is performed, i.e. the detection of antipatterns (see Section 6.1.2) and their solution (see Section 6.1.3).

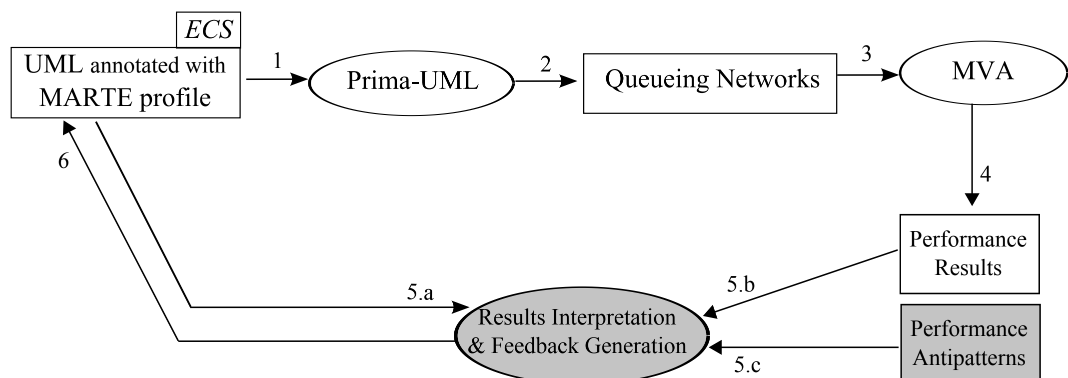


Figure 6.1: ECS case study: customized software performance process.

Figure 6.1 customizes the approach of Figure 1.2 to the specific methodology adopted for this case study in the forward path. The E-commerce System has been modeled with

UML and annotated with the MARTE profile that provides all the information we need for reasoning on performance issues. The transformation from the software architectural model to the performance model is performed with PRIMA-UML, i.e. a methodology that generates a Queueing Network model from UML models [48]. Once the Queueing Network (QN) model is derived, classical QN solution techniques based on well-known methodologies [81] can be applied to solve the model, such as Mean Value Analysis (MVA). The performance model is analyzed to obtain the performance indices of interest (i.e. response time, utilization, throughput, etc.). Thereafter performance antipatterns are used to capture a set of properties in UML models and MARTE profile annotations, thus to determine the causes of performance issues as well as the refactoring actions to overcome such issues.

6.1.1 E-COMMERCE SYSTEM

Figure 6.2 shows an overview of the ECS software system. It is a web-based system that manages business data: customers browse catalogs and make selections of items that need to be purchased; at the same time, suppliers can upload their catalogs, change the prices and the availability of products, etc. The services we analyze here are *browseCatalog* and *makePurchase*. The former can be perfomance-critical because it is required by a large number of (registered and not registered) customers, whereas the latter can be perfomance-critical because it requires several database accesses that can drop the system performance.

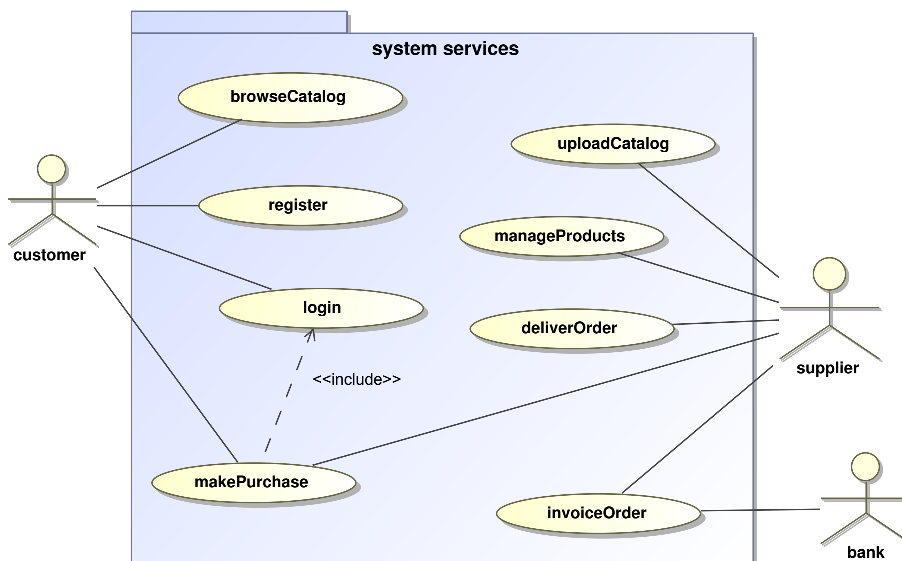


Figure 6.2: ECS case study: Use Case Diagram.

In Figure 6.3 we report an excerpt of the ECS annotated software architectural model. We use UML 2.0 [12] as modeling language and MARTE [13] to annotate additional information for performance analysis (such as workload to the system, service demands, hardware characteristics).

In particular, the UML Component Diagram in Figure 6.3(a) describes the software components and their interconnections, whereas the UML Deployment Diagram of Figure 6.3(b) shows the deployment of the software components on the hardware platform. The deployment is annotated with the characteristics of the hardware nodes to specify CPU attributes (*speedFactor* and *schedPolicy*) and network delay (*blockT*).

Performance requirements are defined for the ECS system on the response time of the main services of the system (i.e. *browseCatalog* and *makePurchase*) under a closed workload with a population of 200 requests/second, and thinking time of 0.01 seconds. The requirements are defined as follows: the *browseCatalog* service must be performed in 1.2 seconds, whereas the *makePurchase* in 2 seconds. These values represent the upper bound for the services they refer to.

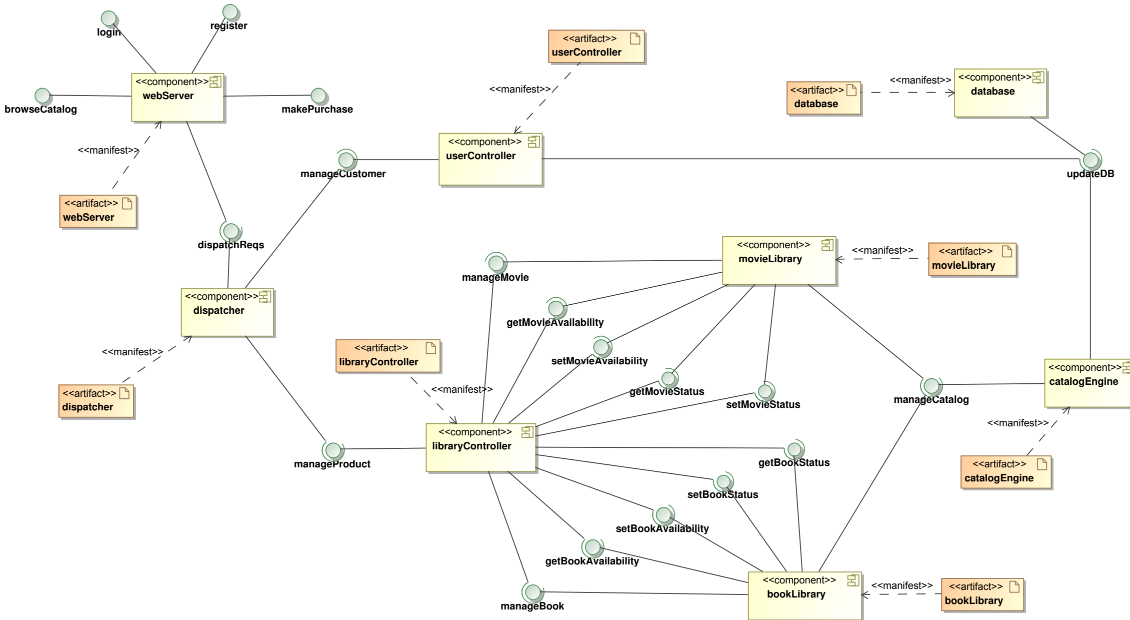
The Prima-UML methodology requires the modeling of: (i) system requirements with a UML Use Case Diagram, (ii) the software dynamics with UML Sequence Diagrams, and (iii) the software-to-hardware mapping with a UML Deployment Diagram. The Use Case Diagram must be annotated with the operational profile, the Sequence Diagram with service demands and message size of each operation, and the Deployment Diagram with the characteristics of hardware nodes (see more details in [48]).

Figure 6.4 shows the Queueing Network model (see more details in Section B.2) produced for the ECS case study. It includes: (i) a set of queueing centers (e.g. *webServerNode*, *libraryNode*, etc.) representing the hardware resources of the system, a set of delay centers (e.g. *wan1*, *wan2*, etc.) representing the network communication delays; (ii) two classes of jobs, i.e. *browseCatalog* (*class A*, denoted with a star symbol in Figure 6.4) is invoked with a probability of 99%, and *makePurchase* (*class B*, denoted with a bullet point in Figure 6.4) is invoked with a probability of 1%.

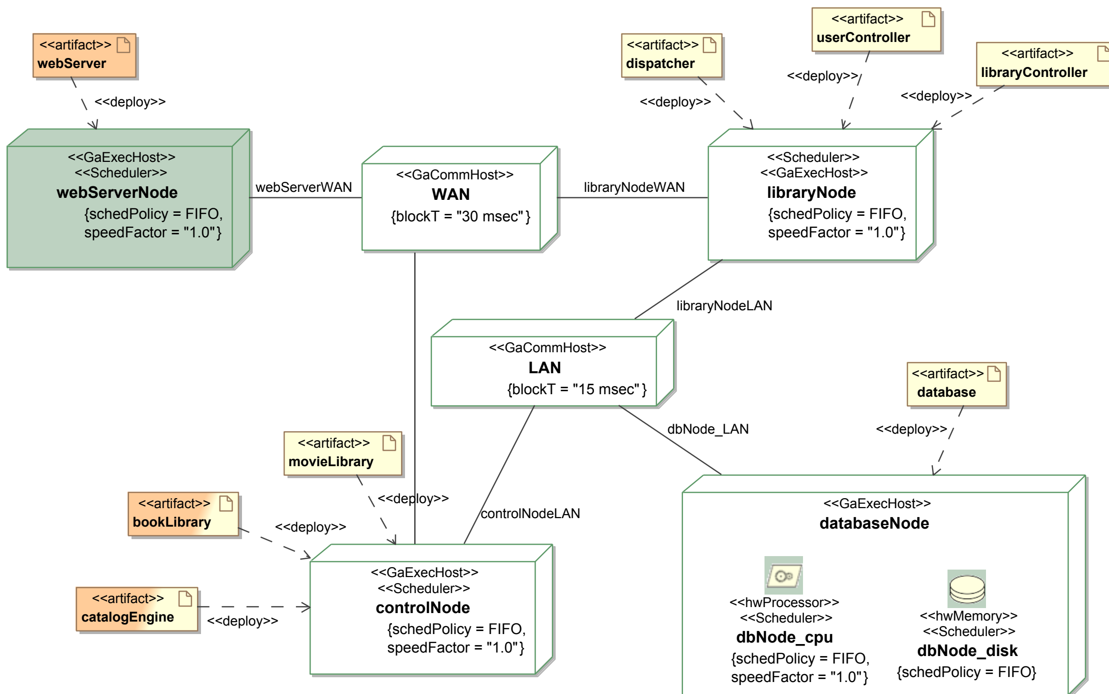
The parametrization of the Queueing Network model for the ECS case study is summarized in Table 6.1. In particular the input parameters of the QN are reported: the first column contains the service center names, the second column shows their corresponding service rates for each class of job (i.e. *class A* and *class B*).

Service Center	Input parameters ECS	
	$class_A$	$class_B$
lan	44 msec	44 msec
wan	208 msec	208 msec
$webServerNode$	2 msec	4 msec
$libraryNode$	7 msec	16 msec
$controlNode$	3 msec	3 msec
db_cpu	15 msec	30 msec
db_disk	30 msec	60 msec

Table 6.1: Input parameters for the queueing network model in the ECS system.



(a) UML Component Diagram.



(b) UML Deployment Diagram.

Figure 6.3: ECS (annotated) software architectural model.

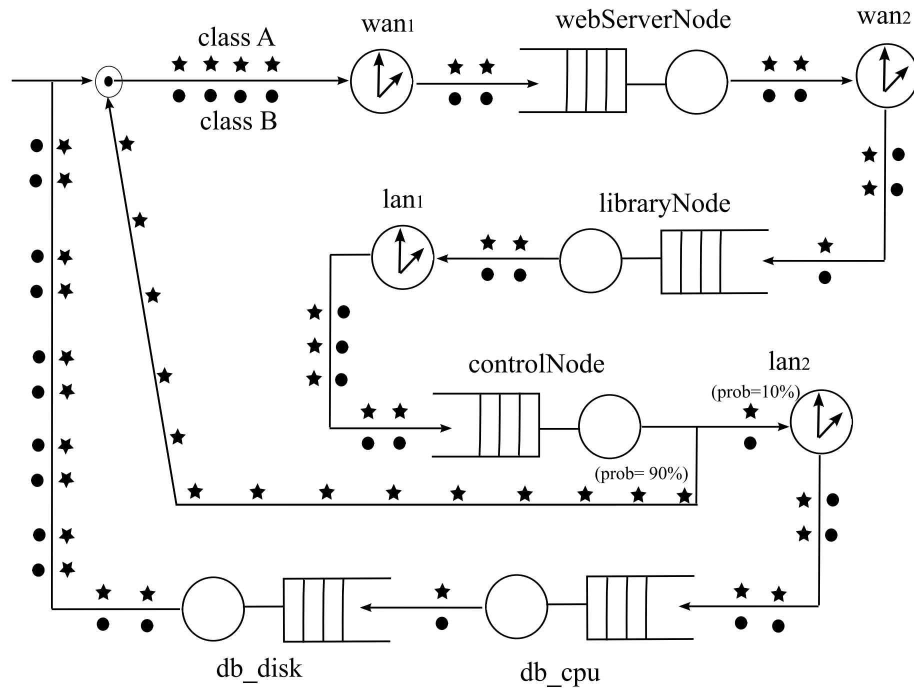


Figure 6.4: ECS - Queueing Network model.

Table 6.2 summarizes the performance analysis results of the ECS Queueing Network model: the first column contains the names of requirements; the second column reports their *required value*s; the third column shows their *predicted value*s, as obtained from the QN solution. As it can be noticed both services have a response time that does not fulfill the required ones: the *browseCatalog* service has been predicted as 1.5 sec, whereas the *makePurchase* service has been predicted as 2.77 sec. Hence we apply our approach to detect performance antipatterns.

Requirement	Required Value	Predicted Value ECS
RT($browseCatalog$)	1.2 sec	1.5 sec
RT($makePurchase$)	2 sec	2.77 sec

Table 6.2: Response time requirements for the ECS software architectural model.

As a first step, the approach joins the (annotated) software architectural model and the performance indices (see Figure 3.1) in an XML representation [5] of the software system.

As said in Section 3.4, basic predicates contain boundaries that need to be actualized on each specific software architectural model. In ECS the estimation of numerical values for thresholds has been calculated as suggested in Tables 3.3 and 3.4.

Table 6.3 reports the binding of the performance *antipatterns boundaries* (see Figure 3.1) for the ECS system. Such values allow to set the basic predicates, thus to proceed with the actual detection.

antipattern	parameter	value
Blob	$Th_{maxConnect}$	4
	$Th_{maxMsgs}$	18
	$Th_{maxHwUtil}$	0.75
	$Th_{maxNetUtil}$	0.85
CPS	$Th_{maxQueue}$	40
	$Th_{cpuMaxUtil}$	0.8
	$Th_{diskMaxUtil}$	0.7
	$Th_{cpuMinUtil}$	0.3
	$Th_{diskMinUtil}$	0.4
EST	$Th_{remMsgs}$	12
	$Th_{remInst}$	5
	$Th_{minNetUtil}$	0.3
...	...	...

Table 6.3: ECS- antipatterns boundaries binding.

6.1.2 DETECTING ANTIPATTERNS

The detection of antipatterns is performed by running the detection engine on the XML representation of the ECS software architectural model . This leaded to assess that: (*lc1*, *bl1*, *browseCatalog*) originates an instance the Blob antipattern; (*libraryNode*, *webServerNode*) originates an instance of the Concurrent Processing Systems antipattern; and finally (*uc1*, *makePurchase*) originates an instance of the Empty Semi Trucks antipattern.

In Figure 6.5 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the parts of the model that evidence the Blob antipattern occurrence. Such antipattern is detected in the ECS software architectural model since there is the instance *lc1* of the component *libraryController* such that (see Table 6.3 and Figure 6.5): (a) it has more than *4* usage dependencies towards the instance *bl1* of the component *bookLibrary*; (b) it sends more than *18* messages (not shown in Figure 6.5 for sake of space); (c) the component instances (i.e. *lc1* and *bl1*) are deployed on different nodes, and the LAN communication host has an utilization (i.e. 0.92), higher than the threshold value (*0.85*).

In Figure 6.6 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the parts of the model that evidence the CPS antipattern occurrence. Such antipattern is detected in the ECS software architectural model since there are two nodes, i.e. *libraryNode* and *webServerNode* that (see Table 6.3 and Figure 6.6) are not used in a well-balanced way. It means that: (i) the queue size of *libraryNode* (i.e. 50) is higher than the threshold value of *40*; (ii) an unbalanced load among CPUs does not occur, because the maximum utilization of CPUs in *libraryNode* (i.e. 0.82 in the $lbNodeproc_1$ instance) is higher than *0.8* threshold value, but the maximum utilization of CPUs in *webServerNode* (i.e. 0.42 in the $wsNodeproc_1$ instance) is not lower than

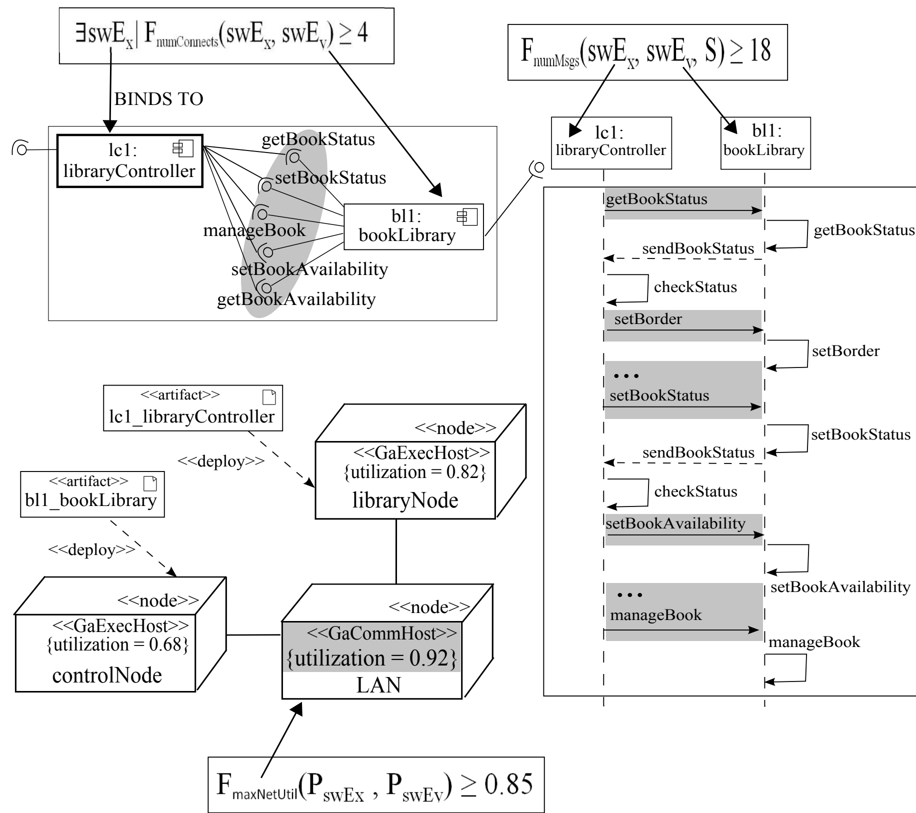


Figure 6.5: ECS- the *Blob* antipattern occurrence.

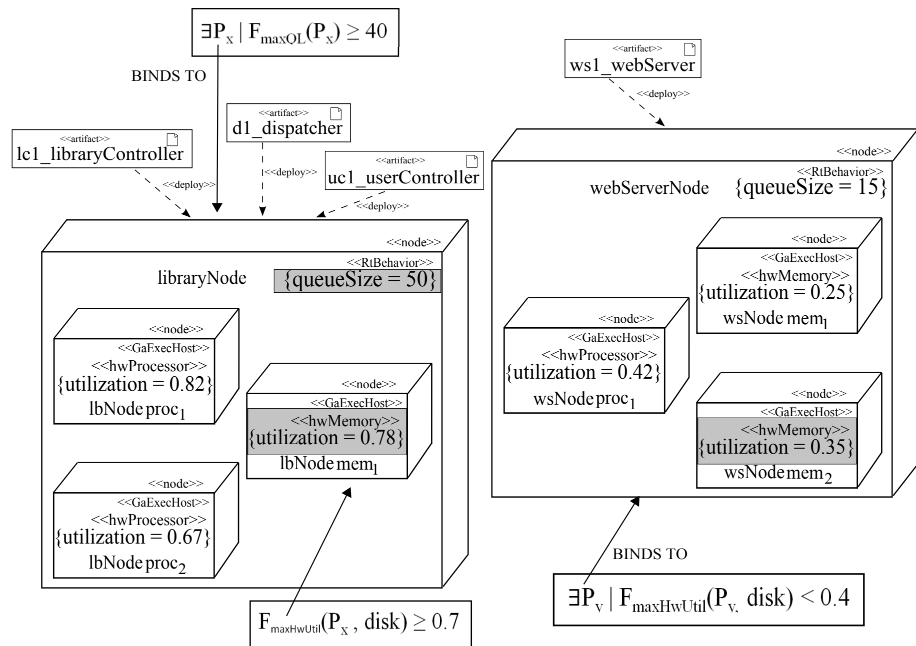


Figure 6.6: ECS- the *Concurrent Processing Systems* antipattern occurrence.

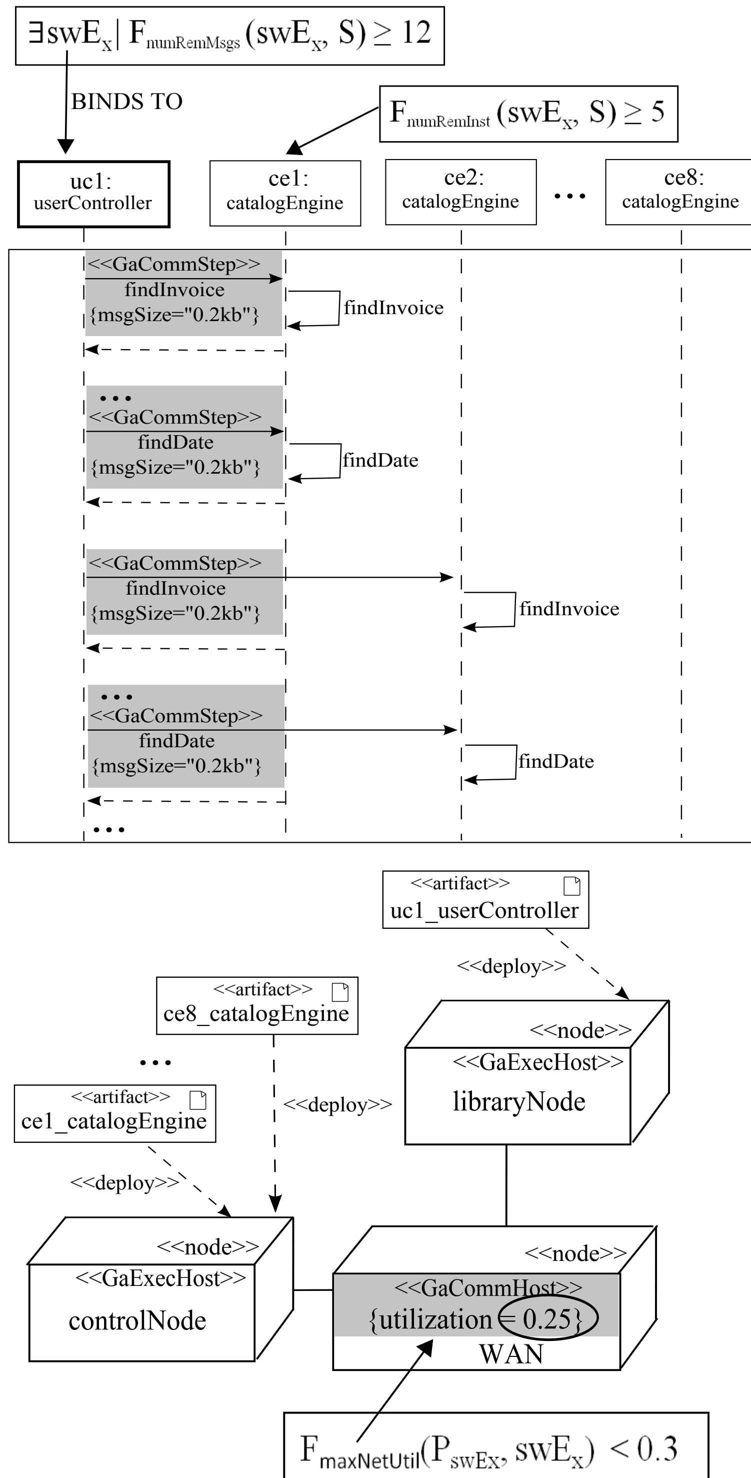


Figure 6.7: ECS- the *Empty Semi Trucks* antipattern occurrence.

Antipattern	Problem	Solution
Blob	*libraryController* performs most of the work, it generates excessive message traffic.	Refactor the design to keep related data and behavior together. Delegate some work from *libraryController* to *bookLibrary*.
Concurrent Processing Systems	Processing cannot make use of the processor *webServerNode*.	Restructure software or change scheduling algorithms between processors *libraryNode* and *webServerNode*.
Empty Semi Trucks	An excessive number of requests is performed for the *makePurchase* service.	Combine items into messages to make better use of available bandwidth.

Table 6.4: ECS Performance Antipatterns: problem and solution.

0.3 threshold value; (iii) an unbalanced load among disks occurs, in fact the maximum utilization of disks in *libraryNode* (i.e. 0.78 in the $lbNodemem_1$ instance), is higher than the threshold value of *0.7*, and the maximum utilization of disks in *webServerNode* (i.e. 0.35 in the $wsNodemem_1$ instance), is lower than the threshold value of *0.4*.

In Figure 6.7 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the parts of the model that evidence the EST antipattern occurrence. Such antipattern occurs since there is the instance *uc1* of the *userController* component such that (see Table 6.3 and Figure 6.7): (a) it sends more than *12* remote messages (not shown in Figure 6.7 for sake of space); (b) the component instances are deployed on different nodes, and the communication host has a utilization (i.e. 0.25 in the wan instance), lower than *0.3* threshold value; (c) it has more than *5* remote instances (*ce1*, ..., *ce8*) of the *catalogEngine* component with which it communicates.

6.1.3 Solving Antipatterns

Whenever an antipattern instance of a certain type has been detected, we tailored the corresponding solution (see Table 3.1) on the ECS system, as reminded in Table 6.4.

According to the antipattern solutions proposed in Table 6.4, we refactored the ECS (annotated) software architectural model and we obtained three new software architectural models, namely $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$, where the Blob, the Concurrent Processing Systems and the Empty Semi Trucks antipatterns have been solved, respectively.

The *Blob* antipattern is solved by modifying the inner behavior of the *libraryController* software component, thus it is not anymore the intermediate component for services provided by the *bookLibrary* and *movieLibrary* components. The *CPS* antipattern is solved by re-deploying the software component *userController* from *libraryNode* to *webServerNode*. The *EST* antipattern is solved by modifying the inner behavior of the *user-*

Controller component in the communication with the *catalogEngine* component for the *makePurchase* service.

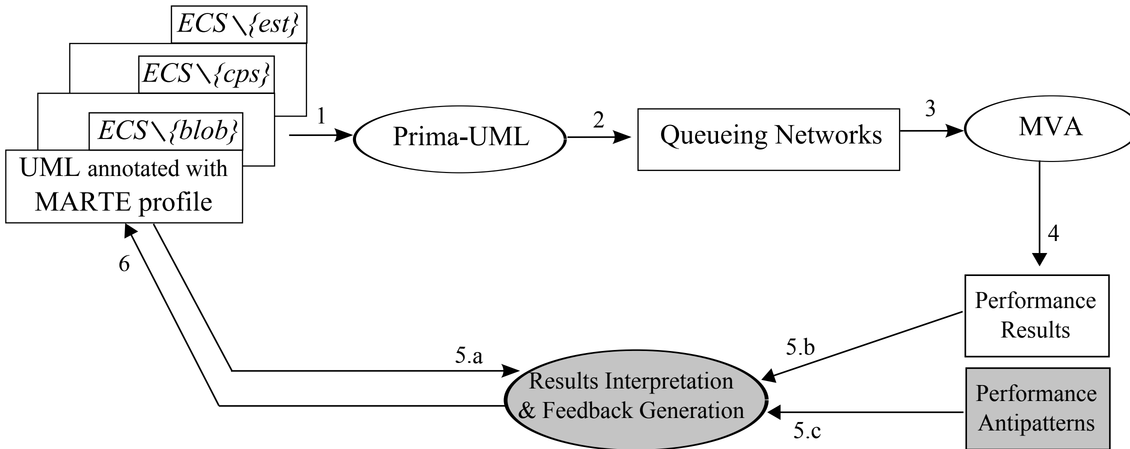


Figure 6.8: ECS model refinement: reiteration of the software performance process.

Service Center	Input parameters					
	$ECS \setminus \{cps\}$		$ECS \setminus \{est\}$		$ECS \setminus \{blob\}$	
	$class_A$	$class_B$	$class_A$	$class_B$	$class_A$	$class_B$
lan	44 msec	44 msec	44 msec	44 msec	44 msec	44 msec
wan	208 msec	208 msec	208 msec	208 msec	208 msec	208 msec
$webServerNode$	**4** msec	**12** msec	2 msec	4 msec	2 msec	4 msec
$libraryNode$	**5** msec	**8** msec	7 msec	**12** msec	**5** msec	**14** msec
$controlNode$	3 msec	3 msec	3 msec	3 msec	3 msec	3 msec
db_cpu	15 msec	30 msec	15 msec	30 msec	15 msec	30 msec
db_disk	30 msec	60 msec	30 msec	60 msec	30 msec	60 msec

Table 6.5: Input parameters for the queueing network model across different software architectural models.

$ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$ systems have been separately analyzed. Input parameters are reported in Table 6.5 where bold numbers represent the changes induced from the solution of the corresponding antipatterns.

For example, in the column $ECS \setminus \{cps\}$ we can note that the service centers *webServerNode* and *libraryNode* have different input values, since the re-deployment of the software component *userController* implies to move the load among the involved resources, in this case from *libraryNode* to *webServerNode*.

In case of *class A*, the load is estimated of 2 msec, in fact in *libraryNode* the initial value of 2 msec in ECS (see Table 6.1) is increased of 2 msec, thus to become 4 msec in $ECS \setminus \{cps\}$ (see Table 6.5), whereas in *webServerNode* the initial value of 7 msec in ECS (see Table 6.1) is decreased of 2 msec, thus to become 5 msec in $ECS \setminus \{cps\}$ (see Table 6.5). In case of *class B*, the load is estimated of 8 msec, in fact in *libraryNode*

the initial value of 4 msec in ECS (see Table 6.1) is increased of 8 msec, thus to become 12 msec in $ECS \setminus \{cps\}$ (see Table 6.5), whereas in *webServerNode* the initial value of 16 msec in ECS (see Table 6.1) is decreased of 8 msec, thus to become 8 msec in $ECS \setminus \{cps\}$ (see Table 6.5).

6.1.4 EXPERIMENTATION

Table 6.6 summarizes the performance analysis results obtained by solving the QN models of the new ECS systems (i.e. $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$ columns), and by comparing them with the results obtained from the analysis of the initial system (i.e. ECS column). The response time of the *browseCatalog* service is 1.14, 1.15, and 1.5 seconds, whereas the response time of the *makePurchase* service is 2.18, 1.6, and 2.24 seconds, across the different reconfigurations of the ECS architectural model.

Requirement	Required Value	Predicted Value			
		ECS	$ECS \setminus \{blob\}$	$ECS \setminus \{cps\}$	$ECS \setminus \{est\}$
RT($browseCatalog$)	1.2 sec	1.5 sec	1.14 sec	1.15 sec	1.5 sec
RT($makePurchase$)	2 sec	2.77 sec	2.18 sec	1.6 sec	2.24 sec

Table 6.6: Response time required and observed.

The solution of the Blob antipattern has satisfied the first requirement, but not the second one. The solution of the Concurrent Processing System leads to satisfy both requirements. Finally, the Empty Semi Trucks solution was useless for the first requirement as no improvement was carried out, but it was quite beneficial for the second one, even if both of them were not fulfilled.

We can conclude that the software architectural model candidate that best fits with user needs is obtained by applying the following refactoring action: the *userController* software component is re-deployed from *libraryNode* to *webServerNode*, i.e. the solution of the Concurrent Processing Systems antipattern. In fact, as shown in Table 6.6 both requirements have been fulfilled by its solution, i.e. the *fulfilment* termination criterion (see Section 1.3). The experimental results are promising, and other decisions can be taken by looking at these results, whereas software architects use to blindly act without this type of information.

This experimentation allows us to ground our antipattern-based process as a general approach that can be applied to the UML modeling notation. Note that the antipattern solution (i.e. the model refactoring) has been manually executed, and it might represent a tedious and error-prone task for the software architect.

6.2 A CASE STUDY IN PCM

First, Section 6.2.1 describes the PCM model of the system under analysis, the so-called Business Reporting System (BRS). Then, the stepwise application of our antipattern-based process is performed, i.e. the detection of antipatterns (see Section 6.2.2) and their solution (see Section 6.2.3). Finally, Section 6.2.4 illustrates a non trivial experimental session.

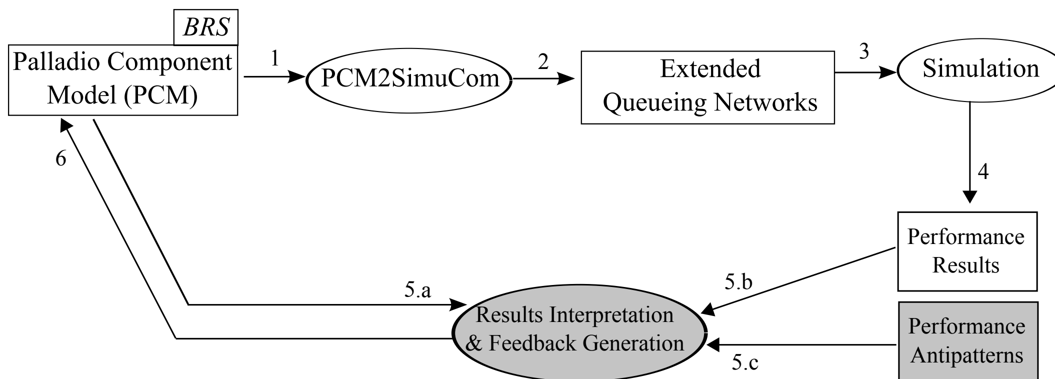


Figure 6.9: BRS case study: customized software performance process.

Figure 6.9 customizes the approach of Figure 1.2 to the specific methodology adopted for this case study in the forward path. The Business Reporting System has been modeled with the Palladio Component Model (PCM). The transformation from the software architectural model to the performance model is performed with PCM2SimuCom, i.e. a methodology that generates an Extended Queueing Network model from PCM models [90]. In particular, it generates the simulation code for the PCM simulation tool SimuCom [22]. The performance model is then simulated to obtain the performance indices of interest (i.e. response time, utilisation, throughput, etc.). Thereafter performance antipatterns are used to capture a set of properties in PCM models, thus to determine the causes of performance issues as well as the refactoring actions to overcome such issues.

The approach has been implemented as an extension to the PCM Bench tool[1] (see Figure 6.10). It allows to automatically interpret the performance analysis results and generate the architectural feedback to the PCM models. Hence, if some performance antipatterns are detected in the PCM model, their solution suggests the architectural alternatives that lead to obtain new software architectural model *candidates* (see Figure 1.4).

Note that the current implementation can detect and solve three antipatterns in PCM models, namely Concurrent Processing Systems, Extensive Processing and One-Lane Bridge. The tool completely automates the described iterative search (see Figure 1.4), supporting the termination criteria of "*no-actions*" and "*#iterations*" (see Section 1.3). In the following we present the results of the experimentation with the PCM Bench tool.

[1]The PCM Bench extension can be downloaded at sdqweb.ipd.kit.edu/wiki/PerOpteryx.

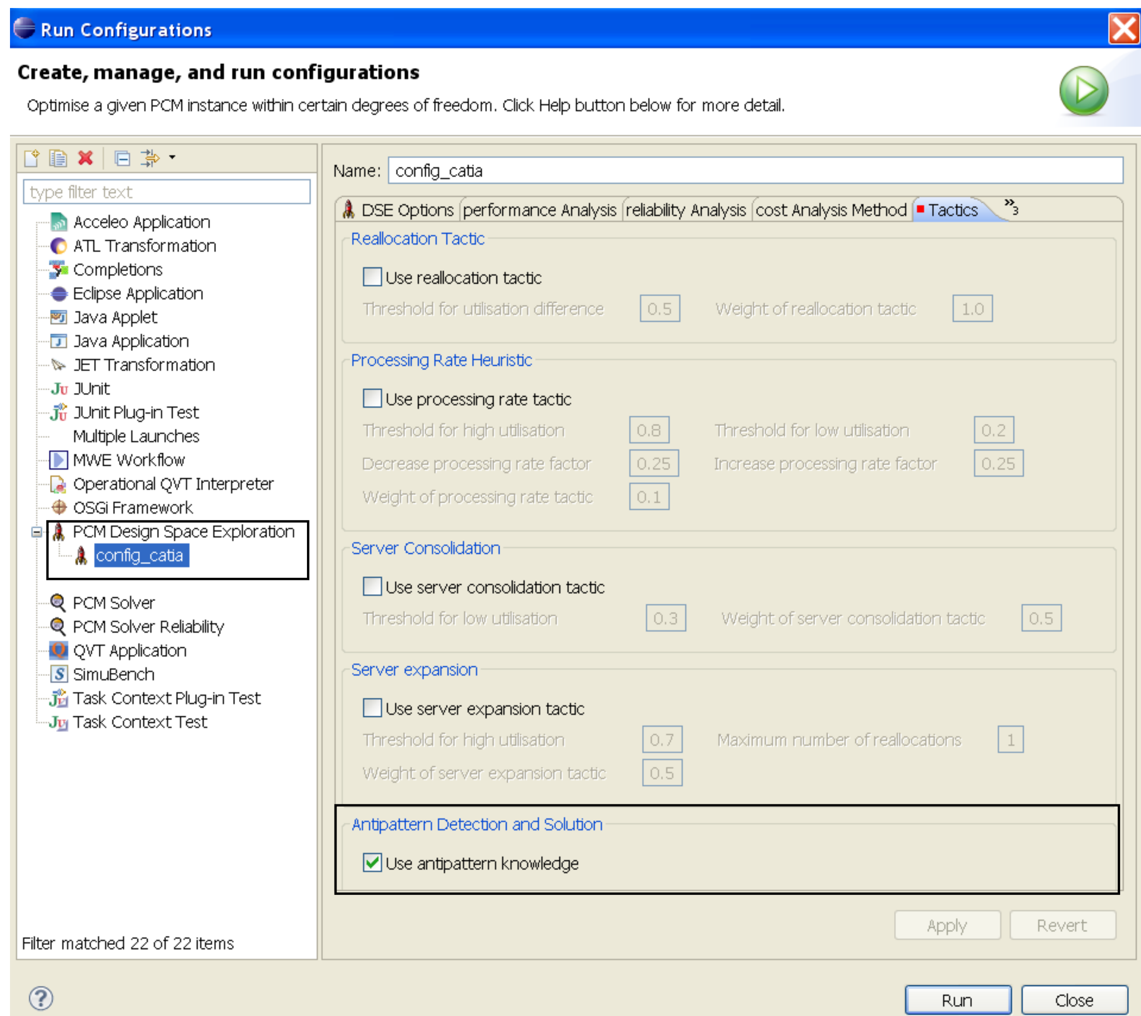


Figure 6.10: Screenshot of the PCM Bench extension providing the usage of antipatterns knowledge.

6.2.1 BUSINESS REPORTING SYSTEM

Figure 6.11 shows an overview of the BRS software system. It lets users retrieve reports and statistical data about running business processes, it is a 4-tier system consisting of several basic components, as described in the following. The *Webserver* handles user requests for generating reports or viewing the plain data logged by the system. It delegates the requests to a *Scheduler*, which in turn forwards the requests. User management functionalities (e.g. login, logout) are directed to the *UserManagement*, whereas report and view requests are forwarded to the *OnlineReporting* or *GraphicalReporting*, depending on the type of request. Both components make use of a *CoreReportingEngine* for the common report generation functionality. The latter one frequently accesses the *Database*, but for some request types uses an intermediate *Cache*. The allocation of software components on resource containers is shown in Figure 6.11, e.g. $Proc_2$ deals with the scheduling of requests by hosting Scheduler, UserManagement, OnlineReporting and GraphicalReporting basic components.

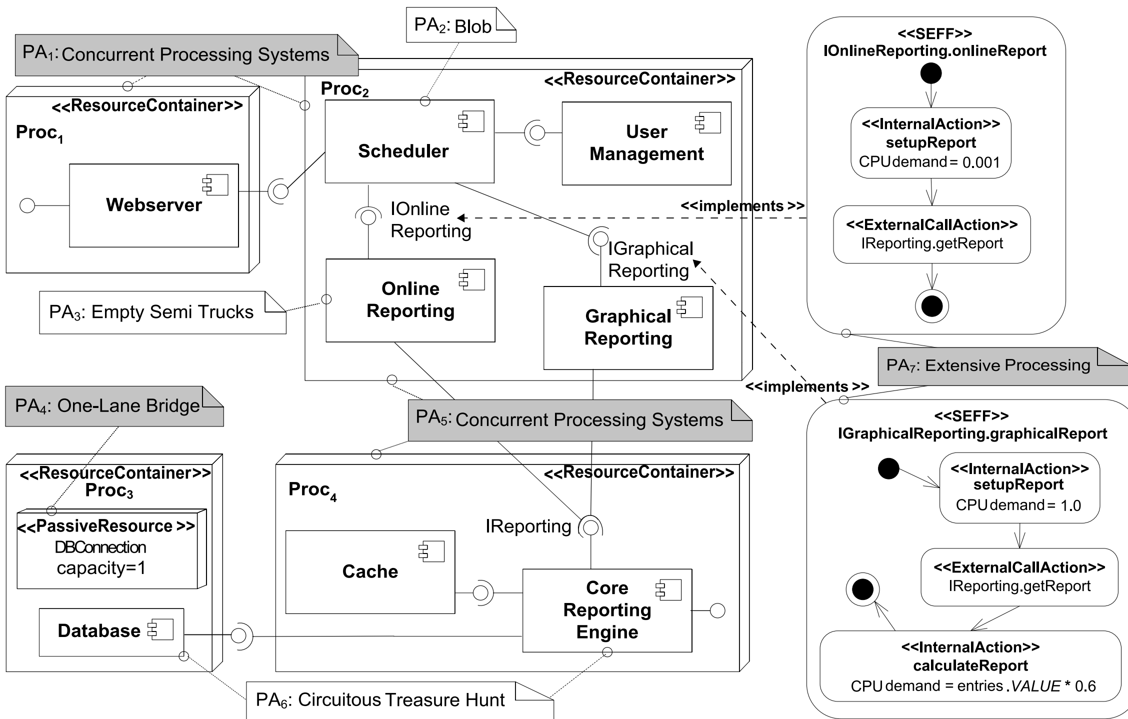


Figure 6.11: PCM software architectural model for the BRS system.

The system supports seven use cases: users can login, logout and request both reports or views, each of which can be both graphical or online; administrators can invoke the maintenance service.

Not all services are inserted in Figure 6.11 for sake of readability, however two examples are shown: *SEFF onlineReport* of component *OnlineReporting* implements the interface *IOnlineReporting*, and *SEFF graphicalReport* of component *GraphicalReporting* implements the interface *IGraphicalReporting*. Both services require an *InternalAction*, that

are performed respectively by OnlineReporting and GraphicalReporting components, to setup the report. Then an *ExternalCallAction* demands to get the report from the CoreReportingEngine component. For the graphicalReport service is necessary to additionally calculate the report for each requested entry. Each internal action is annotated with a resource *demand* indicating the time spent for processing such operation, e.g. the setup of the onlineReport requires 0.001 CPU units.

The PCM software architectural model contains the static structure, the behavior specification of each component and it is annotated with resource demands and resource environment specifications. For performance analysis, the software architectural model is automatically transformed to simulation code, and executed by the SimuCom tool [22].

Figure 6.12 shows the PCM usage model that is the representation of how users use the system: users login, 25 times the onlineView service is invoked, 5 times the graphicalView and onlineReport services are invoked, and finally the graphicalReport and maintain services are performed before the logout.

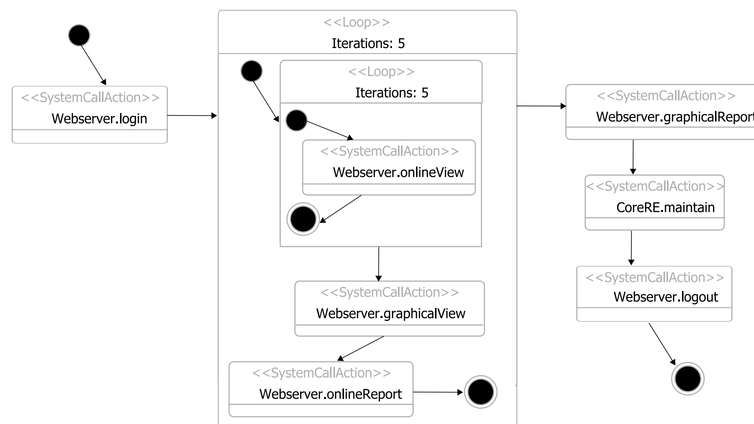


Figure 6.12: PCM usage model for the BRS system.

For sake of simplification, our experimentation is focused on the analysis of the response time of the system, i.e. the average time a user spends in the system according to the defined usage model (see Figure 6.12). The performance analysis of the BRS software architectural model reveals that the response time of the system (under a closed workload of 20 requests with thinking time of 5 seconds) is 18.71 seconds, so it does not meet the required 10 seconds. Since the requirement is not satisfied we apply our approach to detect and solve performance antipatterns.

6.2.2 Detecting Antipatterns

Figure 6.11 shows some labels that indicate the detected antipatterns, because seven instances (PA_1, ..., PA_7) are found. Consider as examples: ($Proc_1$, $Proc_2$) originates an instance of the Concurrent Processing Systems antipattern; (*Scheduler*, *OnlineReporting*,

Antipattern	Problem	Solution
Concurrent Processing Systems	Processing cannot make use of the processor $Proc_1$.	Restructure software or change scheduling algorithms between processors $Proc_1$ and $Proc_2$.
Blob	*Scheduler* performs most of the work, it generates excessive message traffic.	Refactor the design to keep related data and behavior together. Delegate some work from *Scheduler* to *Online Reporting*.
Empty Semi Trucks	An excessive number of requests is performed for the *onlineReport* service.	Combine items into messages to make better use of available bandwidth.
One-Lane Bridge	Processes are delayed while they wait for their turn for accessing the *Database* component.	use the Shared Resources Principle in the *Database* component.
Concurrent Processing Systems	Processing cannot make use of the processor $Proc_4$.	Restructure software or change scheduling algorithms between processors $Proc_2$ and $Proc_4$.
Circuitous Treasure Hunt	An object must look in several places to find the information that it needs in the *Database* component.	Refactor the design to provide alternative paths for accessing the *Database* component.
Extensive Processing	Extensive processing between *onlineReport* and *graphicalReport* services impedes overall response time.	Move extensive processing so that it does not impede high traffic.

Table 6.7: BRS Performance Antipatterns: problem and solution.

onlineReport) originates an instance the Blob antipattern; ($Proc_2$, $Proc_4$) originates an instance of the Concurrent Processing Systems antipattern; (onlineReport, graphicalReport, getReport) originates an instance of the Extensive Processing antipattern. Note that shaded labels represent the antipatterns solvable in the PCM Bench tool, i.e. the only ones that we consider for the solution.

Several instances of the same antipattern can be detected. For example, in the BRS system we found two instances of the Concurrent Processing Systems (see Table 6.7). In this case such antipatterns instances are not independent since they both contain the processor $Proc_2$ as the over utilized one. It is for this reason that we consider refactoring actions separately, similarly to what we have done in the UML example, to avoid unfeasible architectural alternatives.

6.2.3 Solving Antipatterns

Whenever an antipattern instance of a certain type has been detected, we tailored the corresponding solution (see Table 3.1) on the ECS system, as described in Table 6.7.

According to the antipattern solutions proposed in Table 6.7, in the first iteration we refac-

tored the BRS (annotated) software architectural model and we obtained four new software architectural models, namely $BRS \setminus \{PA_1\}$, $BRS \setminus \{PA_4\}$, $BRS \setminus \{PA_5\}$, and $BRS \setminus \{PA_7\}$ (see Figure 6.11) where the Concurrent Processing Systems among $Proc_1$ and $Proc_2$, the One-Lane Bridge, the Concurrent Processing Systems among $Proc_2$ and $Proc_4$, the Extensive Processing antipatterns have been solved, respectively.

The Concurrent Processing Systems antipattern among $Proc_1$ and $Proc_2$ is solved by re-deploying the software component *GraphicalReporting* from $Proc_2$ to $Proc_1$. The One-Lane Bridge antipattern is solved by increasing the capacity of the passive resource *DBconnection* by 5 units. The Concurrent Processing Systems antipattern among $Proc_2$ and $Proc_4$ is solved by re-deploying the software component *GraphicalReporting* from $Proc_2$ to $Proc_4$. The Extensive Processing is solved by changing the scheduling algorithm of $Proc_2$ from First-Come First-Served (i.e. *FCFS*) to *Processor-Sharing*.

6.2.4 EXPERIMENTATION

Figures 6.13 reports our experimentation across multiple *iterations* (see Figure 1.4) of the process: the target performance index is the response time of the *system* and it is plotted on the y-axis, whereas on the x-axis the iterations of the antipattern-based process are listed. Single points represent the response times observed after the separate solution of each performance antipattern.

At the iteration 0 the BRS software architectural model is simulated, and the initial value (18.71 seconds) for the target performance index is reported on the y-axis. As said in Section 6.2.3, in the first iteration we refactored the BRS (annotated) software architectural model and we obtained four new software architectural model candidates. All these four candidates are simulated, and their predicted values for the target performance index are reported in the (1, value) points.

Figure 6.13 summarizes the whole experimentation across *iteration*(s): 40 software architectural model candidates are found, and the response time of the system spans from 18.71 seconds (i.e. the initial value) to 9.26 sec (i.e. the value that fits with the requirement). Note that at each iteration a performance improvement is achieved up to the forth iteration, and the final improvement is roughly of 50%.

Figure 6.14 depicts the process we presented in Figure 1.4 in a graph-like way. Each node reports the performance index of our interest, i.e. RT(system), and its predicted value (e.g. 18.71 seconds in the root of the graph represents the predicted value for the initial system). Each arc represents a refactoring *action* (e.g. the re-deployment of the *GraphicalReporting* component from $Proc_2$ to $Proc_1$) applied to solve a detected *antipattern* (e.g. Concurrent Processing Systems). In our experimentation we applied the *fulfilment criterion* (see Section 1.3) to terminate the process, since the requirement is satisfied at the forth iteration and a software model candidate able to cope with user needs (i.e. the shaded node of Figure 6.14) is found.

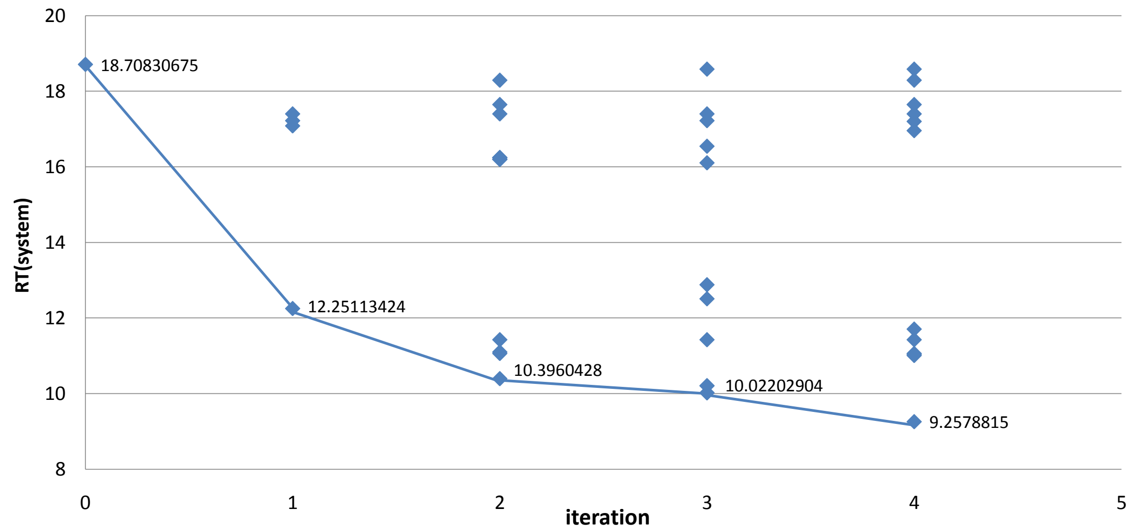


Figure 6.13: Response time of the $system$ across the iterations of the antipattern-based process.

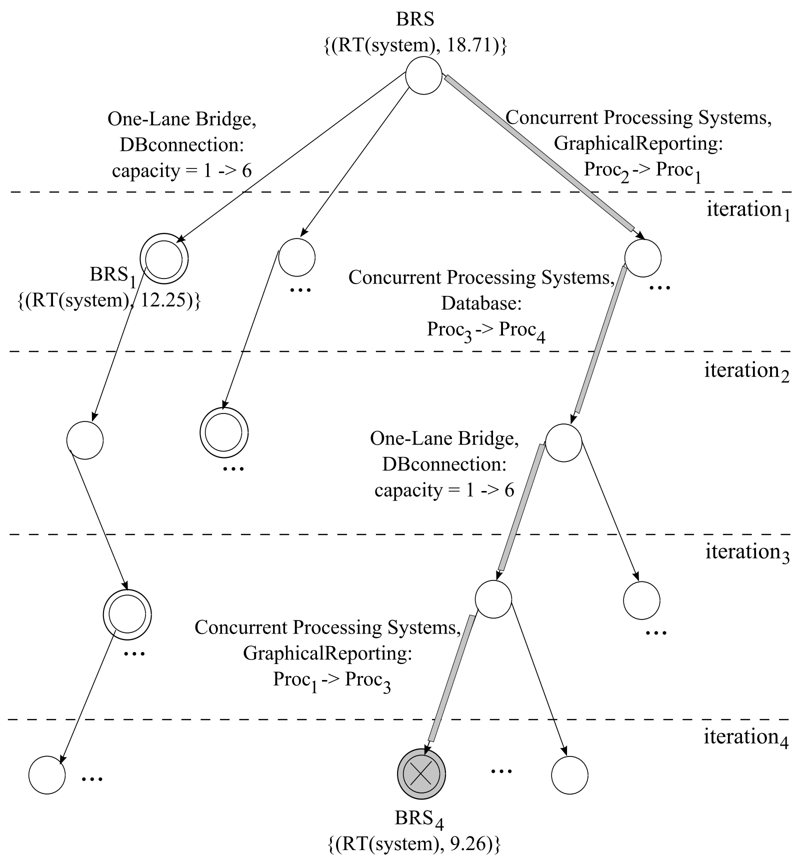


Figure 6.14: Summary of the process for the BRS system.

Figure 6.14 shows at the iteration i a node depicted with a double circle and named BRS_i. Such node is meant to highlight the software architectural model candidate that best fits with the performance index under analysis in the iteration it belongs to. The output candidate is obtained from the fulfillment of a termination criteria, and depicted as a shaded node with a cross. Note that all candidates are newly analyzed and the best candidate of one iteration (local optimum) not necessary is included in the path of the final best candidate (global optimum).

The experimental results we obtained are finally collected in Table 6.8. The performance index of our interest is the response time of the system. Four iterations were performed up to achieve a software architectural model able to cope with the system requirement, and for each iteration i we call BRS_i (see Figure 6.14) the software architectural model candidate that best fits with the requirement under analysis. The first row of Table 6.8 shows the performance improvement of the response time of the system across the different iterations, whereas the second row reports the number of software architectural model candidates that are evaluated at each iteration.

Requirement	Required Value	Observed Value				
		BRS	BRS_1	BRS_2	BRS_3	BRS_4
RT($system$)	10 sec	18.71 sec	12.25 sec	10.40 sec	10.02 sec	9.26 sec
# candidates		1	4	9	11	16

Table 6.8: Response time of the $system$ across BRS software architectural model candidates.

We can conclude that the software architectural model candidate that best fits with user needs is obtained by applying the following refactoring actions (see Figure 6.14): (i) the GraphicalReporting component is re-deployed from $Proc_2$ to $Proc_1$; (ii) the Database component is re-deployed from $Proc_3$ to $Proc_4$; (iii) the capacity of the passive resource DBconnection is increased from 1 to 6; (iv) the GraphicalReporting component is re-deployed from $Proc_1$ to $Proc_3$.

This experimentation allows us to ground our antipattern-based process as a general approach that can be applied to the PCM modeling notation. Note that the antipattern solution (i.e. the model refactoring) has been automatically executed, since it is supported by the PCM Bench tool.

More in general, the step of solving antipatterns opens to multiple problems to be tackled, in fact different cross-cutting concerns (e.g. requirements, workload, operational profile, etc.) might influence the choice of the refactoring actions to apply. Here we like to only say that, once a number of performance antipatterns are detected, a certain strategy has to be introduced to decide which ones have to be solved in order to quickly convey the desired result of users satisfaction (see more details in Chapter 7).

CHAPTER 7

A STEP AHEAD IN THE ANTIPATTERNS SOLUTION

The goal of this Chapter is to present a step ahead in the antipatterns solution, i.e. a technique to decide the most promising model changes that can rapidly lead to remove performance problems. In particular, the antipatterns detected in the software architectural models are ranked on the basis of their guiltiness versus violated requirements. Such ranked list will be the input to the solution step that can use it to give priorities to antipattern solutions.

Without such ranking technique the antipattern solution process can only blindly move among antipattern solutions without eventually achieving the desired result of requirements satisfaction. The core questions tackled in this Chapter are the following: (i) “What are the most guilty antipatterns?” and (ii) “How much does each antipattern contribute to each requirement violation?”.

7.1 A STRATEGY TO IDENTIFY ”GUILTY” PERFORMANCE ANTIPATTERNS

In this Section the problem of identifying, among a set of detected performance antipatterns, the ones that are the real causes of problems (i.e. the “guilty” ones) is tackled. A process to elaborate the performance analysis results and to score performance requirements, model entities and performance antipatterns is introduced. The cross observation of such scores allows to classify the level of guiltiness of each antipattern.

Figure 7.1 reports the process that we propose: the goal is to modify a software architectural model in order to produce a model *candidate* where the performance problems of the former one have been removed. Shaded boxes of Figure 7.1 represent the *ranking step* that is object of this Chapter.

The inputs of the detection engine are: the software architectural model, the performance results, and the performance antipatterns (see Figure 1.2). We here explicitly report performance *requirements* (label 5.d) because they will be used in the ranking step. We obtain two types of outputs from the detection step: (i) a list of *violated requirements*

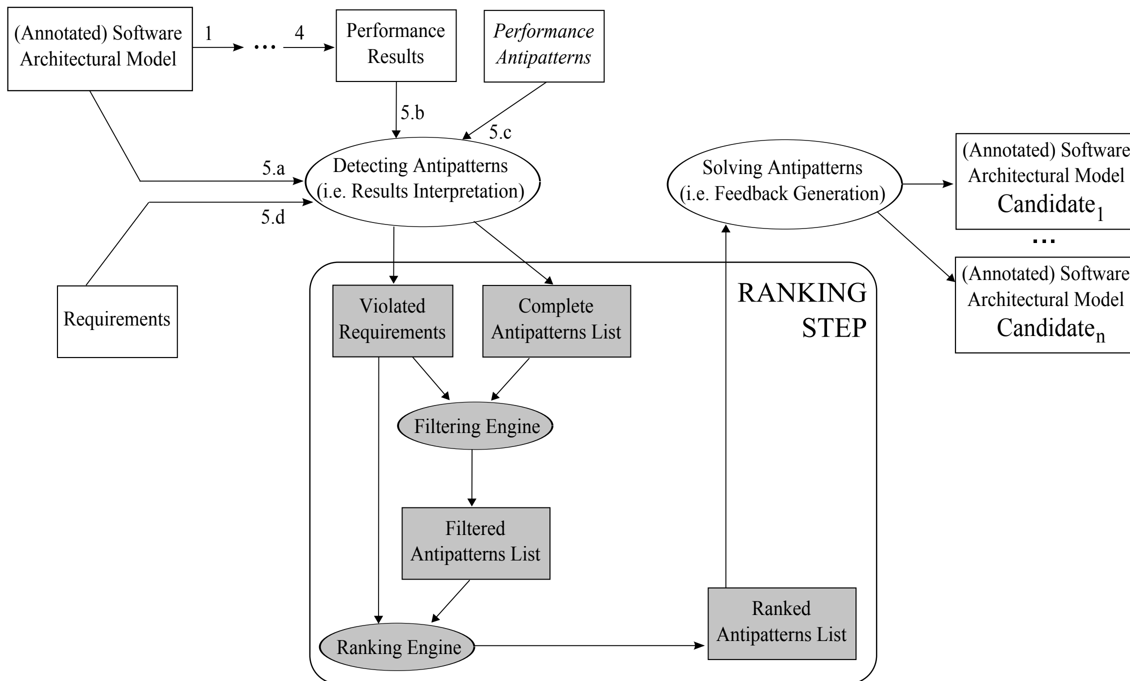


Figure 7.1: A process to improve the performance analysis interpretation.

as resulting from the analysis, and (ii) a *complete antipatterns list*. If no requirement is violated by the current software architectural model then the process terminates here.

Then we compare the complete antipatterns list with the violated requirements and examine relationships between detected antipatterns and each violated requirement through the system entities involved in them. We obtain a *filtered antipatterns list*, where antipatterns that do not affect any violated requirement have been filtered out. In the following step, on the basis of relationships observed before, we estimate how guilty an antipattern is with respect to a violated requirement by calculating a guiltiness score. As a result, we obtain a *ranked antipatterns list* for each violated requirement. Finally, software architectural model *candidate*s can be built by applying to the current software architectural model the solutions of one or more high-ranked antipatterns for each violated requirement.

7.2 AN APPROACH FOR RANKING ANTIPATTERNS

In this section a detailed description of the approach shown in the shaded boxes of Figure 7.1 is provided. The input data for the approach are a set of *violated requirements* (Section 7.2.1) and a *complete antipatterns list* for the system under study (Section 7.2.2). In the first step, we filter out antipatterns that do not affect any requirements and obtain a matrix of *filtered antipatterns* (Section 7.2.3). In the second step, we assign a guiltiness score for the filtered antipatterns with respect to each violated requirement (Section 7.2.4). The resulting *ranked antipatterns list* for each requirement can be used to decide the antipattern solution(s) to apply in order to obtain an improved software architectural model.

7.2.1 VIOLATED REQUIREMENTS

The performance requirements that, upon the model analysis, result to be violated represent very likely the effects (to be removed) of some antipatterns, therefore we focus on them.

System requirements are classified on the basis of the performance indices they address and the level of abstraction they apply.Various levels of abstraction can be defined for a requirement: system, processor, device (e.g., CPU, Disk), device operation (e.g., read, write), software component, basic and composed services. In the following, by "basic service" we denote a functionality that is provided by a component without calling services of other components. By "composed service", we denote a functionality that is provided by a component and involves a combination of calls to services of other components. Both types of services can be offered to the end user at the system boundary, or be internal and only used by other components.

However, we do not consider all possible combinations of indices and levels of abstraction. Our experience on system requirements leads us to focus on the most frequent types of requirements, that concern: *utilization* of processors, *response time* and/or *throughput* of basic and composed services.

Table 7.1 contains simplified examples of performance requirements and their observed values. Each requirement is represented by: (i) an identifier (*ID*), (ii) the type of requirement (*Requirement*) that summarizes the performance index and the target system element, (iii) the required value of the index (*Required Value*), (iv) the maximum system workload for which the requirement must hold (*System Workload*), and (v) the observed value as obtained from the performance analysis (*Observed Value*). In Table 7.1 three example requirements are reported. The first one refers to the utilization index (i.e., U): it requires that processor $Proc_1$ is not utilised more than 70% under a workload of 200 reqs/sec, while it shows an observed utilization of 64%. The second one refers to the response time index (i.e., RT) and the third one refers to the throughput index (i.e., T) of certain software services. Requirements R_2 and R_3 are violated, whereas R_1 is satisfied.

ID	Requirement	Required Value	System Workload	Observed Value
R_1	U($Proc_1$)	0.70	200 $\frac{reqs}{sec}$	0.64
R_2	RT(CS_y)	2 sec	50 $\frac{reqs}{sec}$	3.07 sec
R_3	T(BS_z)	1.9 $\frac{reqs}{sec}$	2 $\frac{reqs}{sec}$	1.8 $\frac{reqs}{sec}$
...	...	...	...	...

Table 7.1: Example of Performance Requirements.

ID	Involved Entities
R_2	$Comp_x.BS_a, Comp_y.BS_b, Proc_2$
R_3	$Comp_w.BS_z, Proc_3$
...	...

Table 7.2: Details of Violated Requirements.

Violated requirements are further detailed by specifying the system entities involved in them. For utilization requirements, we only consider as involved the processor for which the requirement is specified. For example, if a utilization requirement has been specified for processing node $Proc_2$, we consider only $Proc_2$ to be involved. For requirements on services (i.e. response time and throughput requirements), all services that participate in the service provisioning are considered as involved. For example, if a violated requirement is specified for a service S_1, and S_1 calls services S_2 and S_3, we consider all three services S_1, S_2 and S_3 to be involved. Furthermore, all processing nodes hosting the components that provide involved services are considered as involved ([1]). Namely, if the component providing service S_1 is deployed on a processing node $Proc_1$, and the component(s) providing S_2 and S_3 are deployed on $Proc_2$, we additionally consider $Proc_1$ and $Proc_2$ to be involved. With this definition we want to capture the system entities that are most likely to cause the observed performance problems.

In Table 7.2, the involved services of two violated requirements are reported: R_2 involves all basic services participating in the composed service CS_y (i.e., BS_a, BS_b) prefixed by the names of components that provide them (i.e., $Comp_x$, $Comp_y$ respectively), whereas R_3 only involves the target basic service BS_z similarly prefixed. The list of involved entities is completed by the processors hosting these components.

7.2.2 Complete Antipatterns List

We assume that a detection engine has parsed the annotated architectural model and has identified all performance antipatterns occurring in it. All detected performance antipatterns and the involved system entities are collected in a *Complete Antipatterns List*. An example of this list is reported in Table 7.3: each performance antipattern has an identifier (*ID*), the type of antipattern (*Detected Antipattern*), and a set of system entities such as processors, software components, composed and basic services, that are involved in the corresponding antipattern (*Involved Entities*).

[1] The allocation of services to processing nodes is part of the software architectural model.

ID	Detected Antipattern	Involved Entities
PA_1	Blob	$Comp_x$
PA_2	Concurrent Processing Systems	$Proc_1$ $Proc_2$
PA_3	Circuitous Treasure Hunt	$Comp_t.BS_z$
...	...	...

Table 7.3: Complete Antipatterns List.

7.2.3 FILTERING ANTIPATTERNS

The idea behind the step that filters the list of detected antipatterns is very simple. For each violated requirement, only those antipatterns with involved entities in the requirement survive, whereas all other antipatterns can be discarded.

		Requirements			
		R_1	R_2	...	R_j
Antipatterns	PA_1		$Comp_x$		
	PA_2	$Proc_1$			
	...				
	PA_x				e_1, .., e_k

Table 7.4: Filtered Antipatterns List.

A *filtered* list is shown in Table 7.4: rows represent performance antipatterns taken from the complete list (i.e. Table 7.3), and columns represent violated performance requirements (i.e. Table 7.2). A non-empty (x, j) cell denotes that the performance antipattern PA_x is a candidate cause for the violation of the requirement R_j. In particular, the (x, j) cell contains the intersection set of system entities $\{e_1, .., e_k\}$ that are involved in the antipattern PA_x and the violated requirement R_j. We will refer to this set as $involvedIn(PA_x, R_j)$ in the following. Antipatterns that do not have any entity in common with any violated requirement do not appear in this list.

This filtering step allows to reason on a restricted set of candidate antipatterns for each requirement. In Section 7.2.4 we illustrate how to use a filtered antipattern list to introduce a rank for each antipattern that allows to estimate its guiltiness vs. a requirement that has been violated.

7.2.4 Ranking antipatterns

The goal of ranking antipatterns is to introduce an order in the list of filtered antipatterns for each requirement, where highly ranked antipatterns are the most promising causes for the requirement violation. The key factor of our ranking process is to consider the entities involved in a violated requirement. We first assign a score to each entity, and then we rank an antipattern on the basis of a combination of the scores of its involved entities, as follows.

In Table 7.5 we have summarized all equations that we introduce to assign scores to system entities involved in a violated requirement. As outlined in Section 7.2.1, the requirements that we consider are: utilization of processors, response time and throughput of composed and basic services.

Type	Equation
utilization	$score_{i,j} = (observedUtil_i - requiredUtil_j)$
Response time	$score_{i,j} = \frac{ownComputation_i}{maxOwnComputation_j} \cdot \frac{observedRespTime_j - requiredRespTime_j}{observedRespTime_j}$
Throughput	$score_{i,j} = \begin{cases} \frac{requiredThrp_j - observedThrp_j}{requiredThrp_j} & \text{if } workload_i > observedThrp_i \\ & \text{or } isClosed(systemWorkload) \\ 0 & \text{else} \end{cases}$

Table 7.5: How to rank performance antipatterns

Utilization- The violation of an utilization requirement can only target (in this context scope) a processor. For each violated requirement R_j, we introduce a utilization score to the involved processor $Proc_i$ as reported in the first row of Table 7.5. $score_{i,j}$ represents a value between 0 and 1 that indicates how much the $Proc_i$ observed utilization ($observedUtil_i$) is higher than the required one ($requiredUtil_j$).

Response time- The violation of the response time in composed services involves all services participating to that end-user functionality. For each violated requirement R_j, we introduce a response time score to the involved service S_i as reported in the second row of Table 7.5. We quantify how far the observed response time of the composed service CS_j ($observedRespTime_j$) is from the required one ($requiredRespTime_j$). Additionally, in order to increase the guiltiness of services that mostly contribute to the response time of the composed service, we introduce the first multiplicative factor of the equation. We denote with $ownComputation_i$ the observed computation time of a service S_i participating

in the composed service CS_j. If service S_i is a basic service, $ownComputation_i$ equals the response time $RT(S_i)$ of service S_i. However, composite services can also consist of other composite services. Thus, if service S_i is a composite service that calls services S_1 to S_n with probability $P(S_1)$ to $P(S_n)$, $ownComputation_i$ is the response time of service S_i without the weighted response time of called services:

$$ownComputation_i = RT(S_i) - \sum_{1 \leq c \leq n} P(S_c) RT(S_c)$$

We divide by the maximum own computation over all services participating in CS_j, which we denote by $maxOwnComputation_j$. In this way, services with higher response time will be more likely retained responsible for the requirement violation.

The violation of the response time in basic services involves just the referred service. The same equation can be used, where in this case the first multiplicative factor is equal to 1 because $ownComputation_i$ corresponds to $maxOwnComputation_j$.

Throughput- The violation of the throughput in composed services involves all services participating to the end-user functionality. For each violated requirement R_j, we introduce a throughput score to each involved service S_i as reported in the third row of Table 7.5. We distinguish between open and closed workloads here. For an open workload ($isOpen(systemWorkload)$), we can identify bottleneck services S_i that cannot cope with their arriving jobs ($workload_i > observedThrp_i$). A positive score is assigned to these services, whereas all other services are estimated as not guilty for this requirement violation and a score of 0 is assigned to them. For closed workloads ($isClosed(systemWorkload)$), we always observe job flow balance at the steady-state and thus for all services $workload_i$ = $observedThrp_i$ holds. Thus, we cannot easily detect the bottleneck service and we assign a positive score to all involved services. For the positive scores, we quantify how much the observed throughput of the overall composed service ($observedThrp_j$) is far from the required one ($requiredThrp_j$).

The violation of the throughput in basic services involves just this one service. We can use the previous equation as it is, because the only involved service is the one under stress.

Combining the scores of entities Finally, we rank the filtered antipatterns for each violated requirement R_j. To each antipattern PA_x that shares involved entities with a requirement R_j is assigned a guiltiness degree $GD_{PA_x}(R_j)$ that measures the guiltiness of PA_x for R_j. We consider system entities involved in both PA_x and R_j, as reported in the filtered antipatterns matrix $involvedIn(PA_x, R_j)$. We define the guiltiness degree as the sum of the scores of all involved entities:

$$GD_{PA_x}(R_j) = \sum_{i \in involvedIn(PA_x, R_j)} score_{i,j}$$

Thus the problematic entities that have a high score contribute to consistently raise the overall score of the antipatterns they appear in.

7.3 EXPERIMENTING THE APPROACH

In this Section the experimentation of the approach on a business reporting system case study is reported. First, the example software architectural model and the performance analysis are described; then, the stepwise application of the approach is proposed.

7.3.1 CASE STUDY

The system under study is the so-called Business Reporting System (BRS), which lets users retrieve reports and statistical data about running business processes from a data base (see more details in Section 6.2.1). Figure 7.2 shows an overview of the software architectural model, and some labels indicate the detected antipatterns.

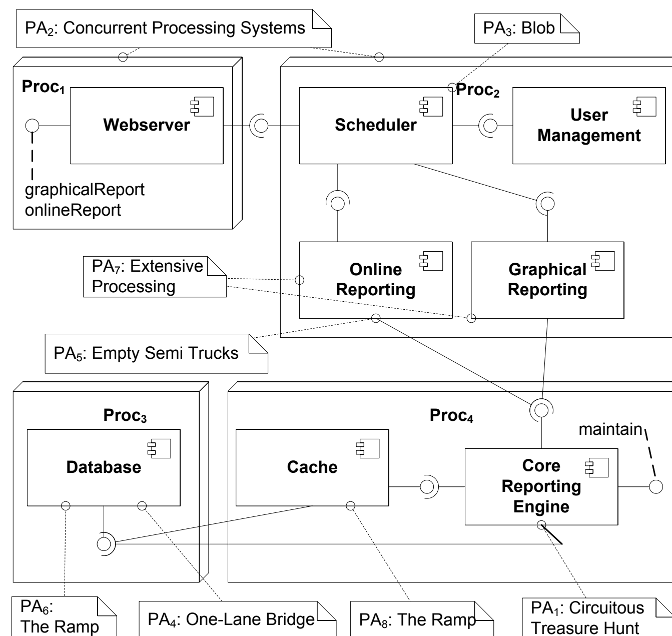


Figure 7.2: BRS software architectural model.

Note that in Figure 7.2 the only depicted services are those ones for which requirements are specified for. Not shown in the diagram is the workload of the system: 30 users use the system in a closed workload with a think time of 10 seconds. The software architectural model of BRS contains the static structure, the behavior specification of each component annotated with resource demands and a resource environment specification. For performance analysis, the software architectural model is transformed automatically

into an Extended Queueing Network model suited for simulation. A discrete-event simulator is used to collect arbitrarily distributed response time, throughput and utilization for all services of the system.

7.3.2 EXPERIMENTAL RESULTS

The results of the performance analysis of the BRS model are reported in Table 7.6, where the focus is on performance requirements and their observed values. The required values are defined according to the end-user expectations, whereas the observed values are the mean values obtained by the simulation. ID's of violated requirements are typed as bold (i.e. R_2, R_5, R_6, R_7, R_9). In the following, the focus will be on solving the shaded R_5 requirement in order to illustrate the approach.

ID	Requirement	Required Value	Observed Value
R_1	U($Proc_1$)	0.50	0.08
$\mathbf{R_2}$	U($Proc_2$)	0.75	0.80
R_3	U($Proc_3$)	0.60	0.32
R_4	U($Proc_4$)	0.40	0.09
$\mathbf{R_5}$	RT($CS_{graphicalReport}$)	2.5 sec	4.55 sec
$\mathbf{R_6}$	T($CS_{graphicalReport}$)	0.5 req/sec	0.42 req/sec
$\mathbf{R_7}$	RT($CS_{onlineReport}$)	2 sec	4.03 sec
R_8	T($CS_{onlineReport}$)	2.5 req/sec	2.12 req/sec
$\mathbf{R_9}$	RT($BS_{maintain}$)	0.1 sec	0.14 sec
R_{10}	T($BS_{maintain}$)	0.3 req/sec	0.41 req/sec

Table 7.6: BRS - Performance requirement analysis

The violated requirements are further detailed with their involved system entities in Table 7.7. Following the approach, the detected performance antipatterns occurring in the software system are collected in the *Complete Antipatterns List*, as shown in Table 7.8. These antipatterns have been also annotated in Figure 7.2 on the architectural model.

The combination of violated requirements and detected antipatterns produces the ranked list of BRS antipatterns shown in Table 7.9. It represents the result of the antipatterns ranking process, where numerical values are calculated according to the equations reported in Table 7.5, whereas empty cells contain a value 0 by default, that is no guiltiness.

ID	Involved Entities
R_2	$Proc_2$
R_5	$WebServer, Scheduler, UserMgmt, GraphicalReport, CoreReportingEngine, Database, Cache$
R_6	$WebServer, Scheduler, UserMgmt, GraphicalReport, CoreReportingEngine, Database, Cache$
R_7	$WebServer, Scheduler, UserMgmt, OnlineReport, CoreReportingEngine, Database, Cache$
R_9	$CoreReportingEngine$

Table 7.7: BRS - Violated Requirements

ID	Detected Antipattern	Involved Entities
PA_1	Circuitous Treasure Hunt	Database.getSmallReport, Database.getBigReport $Proc_3$, CoreReportingEngine.getReport, $Proc_4$
PA_2	Concurrent Processing Systems	$Proc_1$, $Proc_2$
PA_3	Blob	Scheduler, $Proc_2$
PA_4	One-Lane Bridge	Database, $Proc_3$
PA_5	Empty Semi Trucks	OnLineReporting.viewOnLine, $Proc_2$, $Proc_4$ CoreReportingEngine.prepareView, CoreReportingEngine.finishView
PA_6	Ramp	Database, $Proc_3$
PA_7	Extensive Processing	GraphicalReporting, OnLineReporting, $Proc_2$
PA_8	Ramp	Cache, $Proc_4$

Table 7.8: BRS- Complete Antipatterns List

		Requirements				
		R_2	R_5	R_6	R_7	R_9
Anti-patterns	PA_1		0.558	0.122	0.633	
	PA_2	0.054				
	PA_3	0.054	0.051	0.135	0.032	
	PA_4		0.616	0.161	0.689	
	PA_5	0.054				
	PA_6		0.616	0.161	0.689	
	PA_7	0.054	0.125	0.135	0.06	
	PA_8		0.003	0.015	0.03	

Table 7.9: BRS - Ranked Antipatterns List

Table 7.9 can be analyzed by columns or by rows. Firstly, by columns, a certain requirement can be analysed, for example R_5, and then the scores of antipatterns is checked. The approach indicate which antipatterns are more guilty for that requirement violation (i.e., PA_4 and PA_6) and which is the less guilty one (i.e., PA_8). As another example, four antipatterns affect the requirement R_2, but none of them is apparently more guilty than the other ones. So, in this case the approach is able to identify the antipatterns involved without providing a distinction between them. Yet for the requirement R_9 no detected antipattern has a non-zero guiltiness. This means that the violation of R_9 cannot be associated to any known antipattern. In such a case, further performance improvements could be obtained manually, or the requirement has to be relaxed as it is infeasible.

Observing the table by rows, instead, it is possible to distinguish either the antipatterns that most frequently enter the violation of requirements (i.e. PA_3 and PA_7 in this case) or the ones that sum up to the highest total degree of guiltiness (i.e. PA_4 and PA_6 in this case). Different types of analysis can originate from these different views of the ranked list. In the following an analysis by columns on requirements R_5 and R_7 is performed.

In order to satisfy R_5, on the basis of information in Table 7.9 the following antipatterns are separately solved one-by-one: PA_4, PA_6 and, as counterexample, PA_8.

PA_4 is a “One-Lane Bridge” in the `Database`. To solve this antipattern, the level of parallelism in the `Database` is increased, thus at the same time multiple threads can access concurrently. PA_6 is a “Ramp” in the `Database`. Here, the data access algorithms have to be optimised for larger amounts of data. This can be solved with a reduced resource demand of the database. In the example, the assumption is that the resource demand is halved. PA_8 is a “Ramp” in the `Cache`. The latter accumulates more and more

information over time and is slowed down. This can be solved with a reduced resource demand of the `Cache`. In the example, the assumption is again that the resource demand is halved.

The results of the ***new*** software systems (i.e., BRS_{PA_x}, the BRS initial system with PA_x solved) are collected in Table 7.10. It can be noticed that the high guiltness degrees of PA_4 and PA_6 have provided a relevant information because their removal consistently improved the response time. After the removal of PA_8, instead, the requirement R_5 is still violated because it has been removed a cause that affects much less the violated requirement considered.

ID	Requirement	Required Value	Observed Value			
			BRS	BRS_{PA_4}	BRS_{PA_6}	BRS_{PA_8}
R_5	RT($CS_{graphicalReport}$)	2.5 sec	4.55 sec	2.14 sec	2.06 sec	4.73 sec

Table 7.10: RT($CS_{graphicalReport}$) across different software architectural models

In Figure 7.3 the experiments on the requirement R_5 are summarized. The target performance index of R_5 (i.e. the response time of the *graphicalReport* service) is plotted on the y-axis, whereas on the x-axis the degree of guiltiness of antipatterns is represented. The horizontal bottommost line is the requirement threshold, that is the response time required, whereas the horizontal topmost line is the observed value for the original BRS system before any modification. Single points represent the response times observed after the separate solution of each performance antipatterns, and they are labeled with the ID of the antipattern that has been solved for that specific point. Of course, the points are situated, along the x-axis, on the corresponding guiltiness degree of the specific antipattern.

What is expected to observe in such representation is that the points approach (and possibly go below) the required response time while increasing their guiltiness degree, that is while moving from left to right on the diagram. This would confirm that solving a more guilty antipattern helps much more than solving a less guilty one, thus validating the guiltiness metric.

All antipatterns with non-zero guiltiness have been solved, one by one, to study their influence on the requirement R_5. Figure 7.3 very nicely validates the hypothesis, in that very guilty antipatterns more dramatically affect the response time, and their solution leads towards the requirement satisfaction. The same considerations made above can be reiterated for the other requirements.

For example, Figures 7.4 and 7.5 respectively represent the experiments on the requirements R_6 (i.e. the throughput of the *graphicalReport* service) and R_7 (i.e. the response time of the *onlineReport* service).

In Figure 7.4 we can observe that the antipatterns that have been assigned a significant guiltiness score improve the system's response time for that service. The score reflects

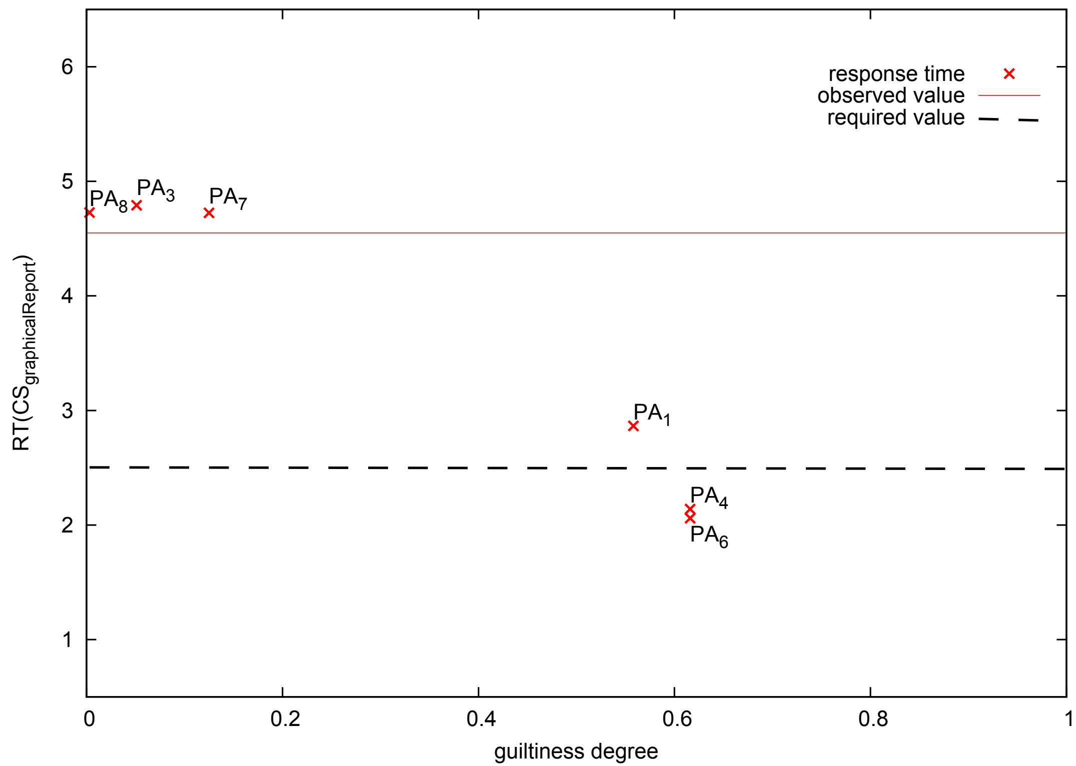


Figure 7.3: RT($CS_{graphicalReport}$) vs the guiltness degree of antipatterns.

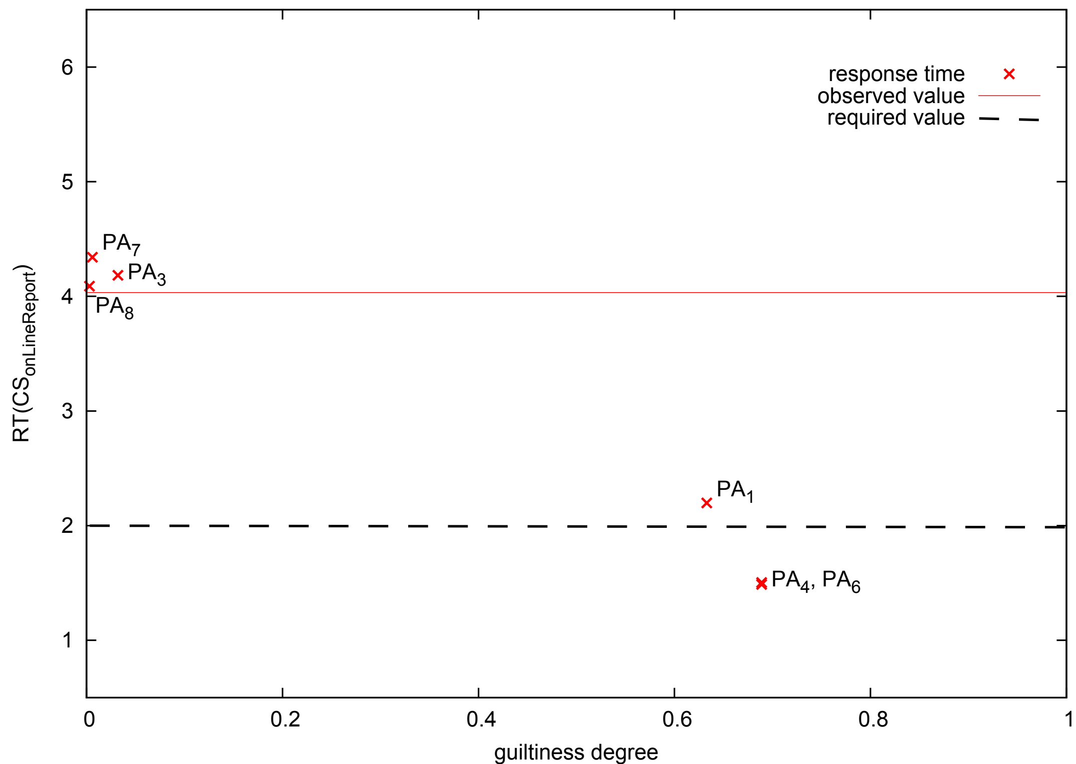


Figure 7.4: RT($CS_{onlineReport}$) vs the guiltness degree of antipatterns.

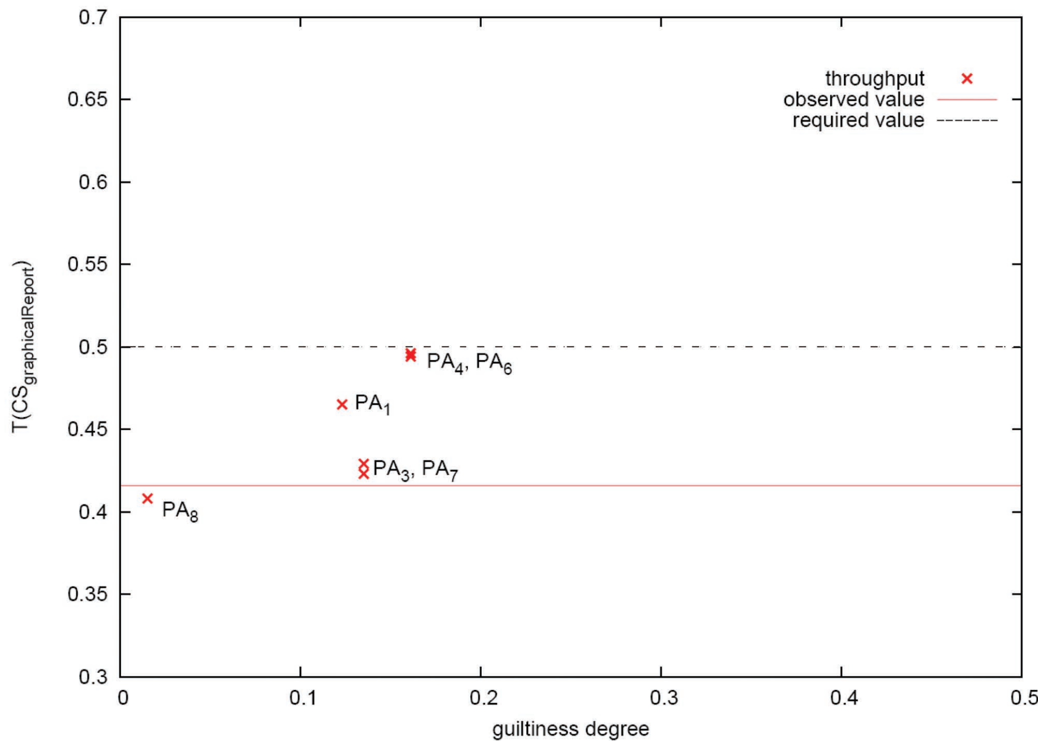


Figure 7.5: T($CS_{graphicalReport}$) vs the guiltness degree of antipatterns.

the potential improvement of each antipattern as well.

In Figure 7.5 we can observe that the solution of the antipatterns that have been assigned a significant guiltiness score (i.e. PA_4 and PA_6) indeed improve the system throughput, by increasing the initial value. The score can roughly predict the amount of improvement: the solution of antipatterns PA_3 and PA_7 is more beneficial in comparison to the PA_1 antipattern. Still, none of the antipattern solutions can ultimately satisfy the requirement.

7.4 Discussion

The experimentation in Section 7.3 shows promising results for the architectural model example and the types of requirements introduced. This is a proof of concept that such a ranking approach can help to identify the causes of performance problems in a software system. The experimentation phase has been very important to refine the approach, in fact by observing the performance analysis results the equations that represent the antipattern ranking have been fine tuned.

However, this is only a first step in this direction, and several issues are yet to be addressed. Although promising results have been obtained in the experimentation, the score model can certainly be improved and needs more experimentation on models of different application domains.

Other types of requirements, among the one listed in Section 7.2, may need appropriate formulas for scoring the entities involved in them. Nested requirements could be preprocessed to eliminate from the list of violated requirements those that are dominated from other ones.

More experience could lead to refine the antipattern scoring on the basis of, let say, the application domain (e.g. web-based application) or the adopted technology (e.g. Oracle DBMS). For example, a detected “Circuitous Treasure Hunt” might be of particular interest in database-intensive applications, whereas a detected “Concurrent Processing Systems” might be more important for web-based applications.

Finally, to achieve more differentiation in the scoring process for guilty performance antipatterns, negative scores to the entities involved in satisfied requirements can be devised.

The benefit of using the ranked antipattern list of antipatterns is to decide the most promising model changes that can rapidly lead to remove performance problems. In this direction several interesting issues have to be faced, such as the simultaneous solution of multiple antipatterns. In the following we report some ideas about how to apply *moves*, i.e. the solution of a set of performance antipatterns at the same time.

7.5 Towards the simultaneous solution of antipatterns

A graphical representation on how to perform the combination of performance antipatterns solutions is shown in Figure 7.6. Starting from the ranked antipatterns list, it is possible to plan a set of different *moves:* $\{M_1$, ..., $M_k\}$, each containing a set of some performance antipatterns: $M_i = \{PA_x, \ldots, PA_y\}$. The application of the move M_k on a software architectural model aims at obtaining another software architectural model candidate, where a set of antipatterns $PA_x, \ldots, PA_y$ have been solved.

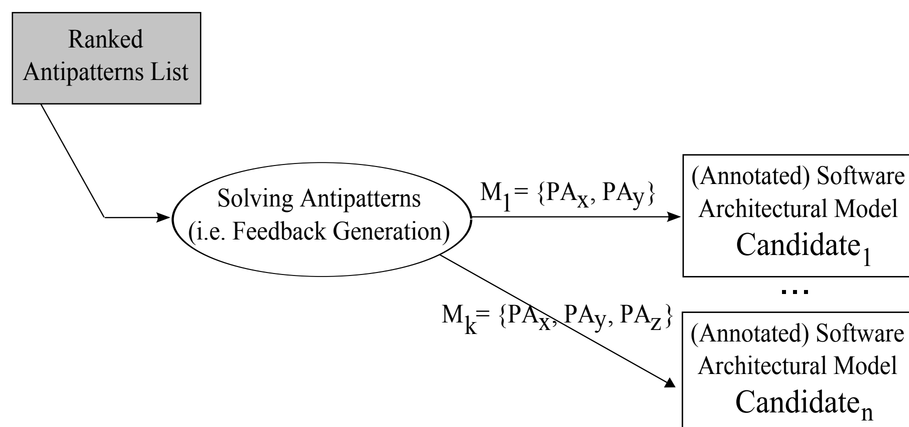


Figure 7.6: How to decide between different moves.

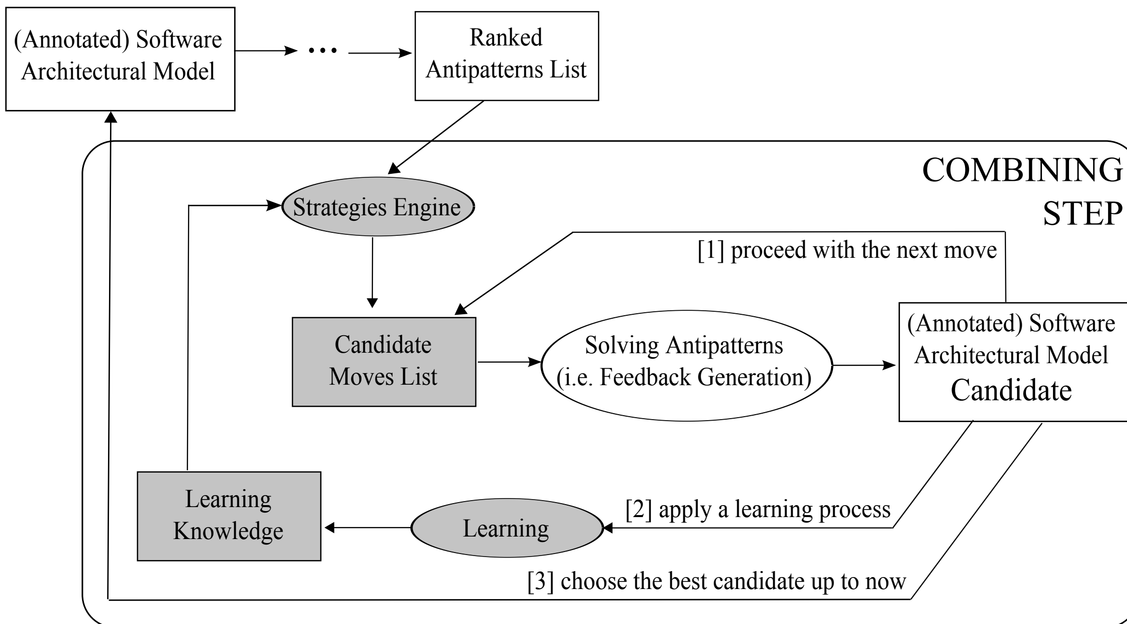


Figure 7.7: Process for the combination of performance antipatterns.

Applying moves In Figure 7.7 we report a process able to manage the combination of the performance antipatterns. The process shown in Figure 7.7 is the continuation of the Figure 7.1, in fact we report the starting point (i.e. *annotated software architectural model*) and the result of the ranking antipatterns process (i.e. *ranked antipatterns list*). Thus the dots of Figure 7.7 represent the entire process of Figure 7.1.

The goal of the *combining step* (see Figure 7.7) is to define a *candidate moves list*, i.e. a list of moves $\{M_1, \ldots, M_k\}$, ordered from the most promising one. It means that a ranking process for moves must be introduced. It is also possible to define some strategies to further reorder the list if we observe particular properties of the moves.

After the application of a move we obtain a software architectural model *candidate* to be evaluated, and a set of new performance results on the latter one are obtained. If all the requirements are satisfied the process can be stopped, we have found a candidate able to solve the performance flaws. If the new predicted values do not fit with all the requirements, it is possible to apply three different options, as depicted in Figure 7.7:

[1] coming back to the candidate moves list and applying the next one, as previously defined;

[2] coming back to the set of candidate moves, but possibly re-ordering the list of candidate moves by means of a *learning* step based on the moves already performed, thus to extract the *learning knowledge* able to support such re-ordering;

[3] coming back to apply the whole process of ranking antipatterns from the current software architectural model candidate.

Ranking moves In the following we collect some ideas on how to evaluate a combination of performance antipattern solutions, thus to obtain a rank on the moves. Six metrics, i.e. $Mt_1, \ldots, Mt_6$, can be devised to evaluate the candidate moves. In the following, we define

$$involvedInMove(M_i) := \bigcup_{PA_x \in M_i} involvedInPA(PA_x)$$

to be the set of all entities addressed by the antipattern PA_x of the move M_i.

Mt_1: *Coverage of the model*- we count the number of entities, involved in the requirement violations, that are covered by the suggested move, i.e. by the antipatterns the move M_i consists of.

$$coverageModel_i = |involvedInMove(M_i)|$$

Mt_2: *Score of the involved entities*- we sum up the scores of all involved entities addressed in move M_i:

$$scoreMove_i = \sum_{e \in involvedInMove(M_i)} score_x$$

Mt_3: *Cost of the move*- we can introduce a parameterized cost function for each type of antipattern. For example, the god class antipattern type might have a cost function

$$cost = number_of_messages \cdot 5 + number_of_associations \cdot 2$$

With this functions, we can determine $cost(PA_x)$ as the cost of a particular antipattern PA_x. Then, we obtain the overall cost of a move as

$$cost_i = \sum_{PA_x \in M_i} cost(PA_x)$$

Mt_4: *Scores of the antipatterns*- we can take into account whether a move focuses on promising antipatterns, i.e. antipatterns with a high score. Thus, we include the score of the antipatterns in the move M_i:

$$scoreAP_i = \sum_{PA_x \in M_i} score_x$$

Mt_5: *Number of addressed antipatterns*- in order to prefer moves that address few, but important antipatterns, we also include the total number of antipatterns applied in a move M_i. Thus, we can derive the average score of the antipatterns involved with this metric and Mt_4:

$$NumberOfAP_i = |\{PA_x \in M_i\}|$$

Mt_6: *Coverage of the requirements*- the final goal is to solve all requirements, hence for each violated requirement R_j, at least one of the involved entities $involvedInReq(R_j)$ or affected entities $affectingReq(R_j)$ has to be covered by the move. We can count the number of requirements addressed by move M_i. With weighting factors ($involvedFactor$, $affectedFactor$) we allow to weight the importance of requirements covered basing on involved entities (the requirements are collected in the set $setCovReqInvolved$, see below) and requirements only covered by affected entities (which can be considered less covered):

$$setCovReqInv_i = \left\{ r \;\middle|\; \begin{array}{l} r \textbf{ is a violated requirement} \\ \wedge\, invInReq(r) \cap invInMove(M_i) \neq \emptyset \end{array} \right\} \quad (7.1)$$

$$covInv_i = |setCovReqInv_i| \quad (7.2)$$

$$covAff_i = \left\{ r \;\middle|\; \begin{array}{l} r \textbf{ is a violated requirement} \\ \wedge\, r \in setCovReqInv_i \\ \wedge\, affReq(R_j) \cap invInMove(M_i) \neq \emptyset \end{array} \right\} \quad (7.3)$$

$$coverageReqs_i = invFactor \cdot covInv_i + affFactor \cdot covAff_i \quad (7.4)$$

Note that the actual decision of which metric is more or less important may depend on the actual software system under study and its domain. In addition to the metrics, several strategies might allow to re-order the promising moves; we adopt the following syntax for strategies, i.e. $Strategy_x$ $[condition_y] - > actions : \{action_z, ...\}$.

For example: $Strategy_1$ [performance antipatterns deal with the same entities involved] $- > actions$: {apply the one with the highest value of the score, and introducing the others only after}. This means that if PA_i has a lower value of the score than PA_j, PA_i is only used as a refinement of PA_j. All moves that contain PA_i without PA_j are put at the end of the list, or they can be punished with a negative score.

Learning In the following we collect some ideas on how to perform a learning process from the performance antipattern solutions, thus to obtain knowledge about the effects of single antipatterns and their combinations for the software architectural model under study and for the new candidates derived from it.

Two methodologies can be used to perform the learning activity. The first methodology is that after evaluating several moves, we might detect that a certain combination of antipatterns (e.g. the pair PA_1, PA_5) is correlated with a particularly good or bad performance. Based on such observation, we can add a positive or negative extra score to all moves containing this combination. The second methodology is that the weights of the different metrics to rank the moves could be adjusted based on the evaluation of a number of moves, which may lead to a re-ordering the list of moves. For example, it could be detected that moves with a high score for the coverage of the model exhibited rather poor performance indices, hence it is better to reduce the weight for this score.

CHAPTER 8

FURTHER STEPS IN THE ANTIPATTERNS SOLUTION

In this Chapter we discuss the cross-cutting concerns that influence the automated generation of architectural feedback in order to give a wide explanation of issues that might emerge in this context. To this scope we define a set of research directions aimed at: (i) verifying to which extent the architectural feedback can be defined; (ii) showing the challenges and the areas where further research is required.

Similarly to Chapter 7 in which the analysis of the requirements is used to support the antipatterns solution process, other emerging research directions are outlined: the workload sensitivity, the stakeholders, the operational profile, and the cost analysis. All these activities are meant to support the antipatterns solution by providing a strategy to decide which ones have to be solved in order to quickly convey the desired result of users satisfaction.

8.1 WORKLOAD SENSITIVITY ANALYSIS

The workload sensitivity analysis represents the process of evaluating the relationship between individual or groups of requests and their demands. In general, it is quite common to focus on the expected workload for a specific system even if workloads can vary in many different situations and maybe the analysis of peaks can be useful to stress the system up to point out its limitations.

Consider the following example. A performance model predicts that a service, e.g. S_x, has a response time of 3.5 seconds under a workload of 100 requests/second. What happens while increasing the number of requests? The designer is expecting the maximum workload able to fulfill the required value. If the requirement is that the service S_x must have a response time not greater than 5 seconds than the designer is interested to know that the required value is achieved up to a certain workload, e.g. of 300 requests/second.

Figure 8.1 shows how the workload sensitivity analysis activity can be used to prioritize performance antipatterns: on the x-axis the workload, expressed in requests/second, is reported; on the y-axis the performance antipatterns instances are listed. A "×" symbol is in the ($workload_value$, PA_i) point if a performance antipattern PA_i occurs when the workload is fixed to that $workload_value$.

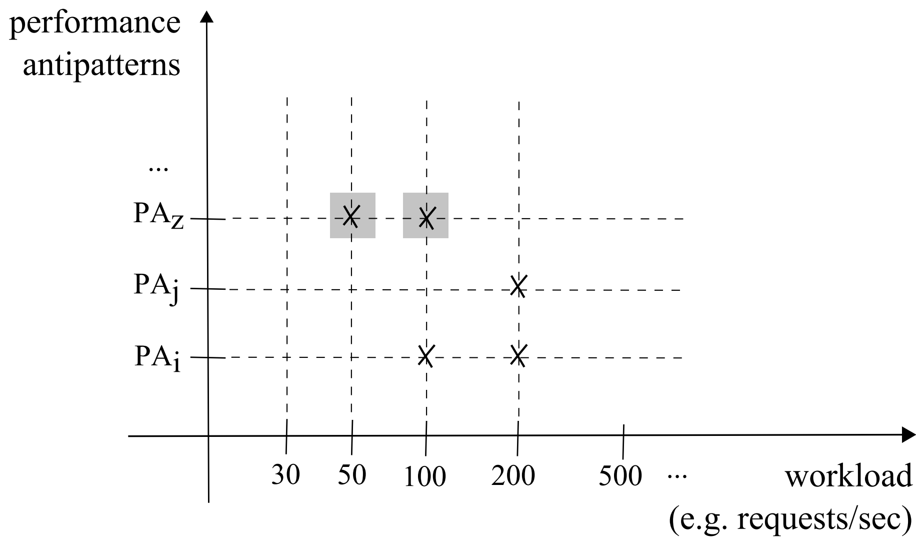


Figure 8.1: Workload sensitivity analysis as a support for solving antipatterns.

For example, in Figure 8.1 we can notice that if the workload is 30 requests/second no antipatterns occur, hence there should be no performance problems in the system. Instead a workload of 50 requests/second leads the occurrence of the PA_z antipattern. Note that a further increase of the workload usually makes worse the system performance. We might expect that other antipatterns will be triggered, e.g. a workload of 100 requests/second adds the occurrence of the PA_i antipattern, whereas a workload of 200 requests/second may cause the disappearance of the PA_z antipattern.

The mapping between performance antipatterns and workload may be useful to give a priority in the sequential solution of antipatterns. For example, under a workload of 100 requests/second PA_z and PA_i antipatterns are detected, but PA_z may be considered more critical than PA_i whose first occurrence appears with a higher value of the workload (i.e. it may be considered as less critical).

In the context of the workload sensitivity analysis it might be useful to devise methodologies to examine if performance scenarios have a different workload intensity for different situations such as peak hour, average hour, peak day, and so on. In fact it is fundamental to have an estimate of the workload intensity along the time, since it provides a basis for evaluating the amount of work under which the system operates, thus to estimate the effectiveness of the final software architectural model [122].

8.2 Stakeholders analysis

Stakeholders are persons or organizations (e.g. legal entities such as companies) that have an interest in the software system, and they may be affected by it either directly or indirectly. The term "stakeholder" can include several meanings such as it might refers

to someone who operates in the system (normal and maintenance operators) or someone who benefits from the system (functional, political, financial and social beneficiaries) or someone involved in purchasing or procuring the system (mass-market customers) [125]. Stakeholder analysis refers to the process of analyzing the attitudes of stakeholders and such evaluation can be done one shot or on a regular basis to track how stakeholders changed their attitudes over time.

Consider the following example. There are two different stakeholders: the first one is a customer and requires to buy products more and more faster, e.g. the service $makePurchase$ must have a response time not larger than 2 seconds; the second one is a supplier of the products and it requires to sell products only if they are already available because the production of old products might be too expensive. The service $makePurchase$ becomes a critical one, since for each request it is necessary to check the quantity of the products in the database and the response time of the service is predicted to 3 seconds. Database accesses are considered critical and they must be improved.

Figure 8.2 shows how the stakeholders analysis activity can be used to prioritize performance antipatterns: on the x-axis the different stakeholders are reported; on the y-axis the performance antipatterns instances are listed. A "×" symbol is in the ($stakeholder_x$, PA_i) point if a performance antipattern PA_i occurs when considering the requirements of the $stakeholder_x$.

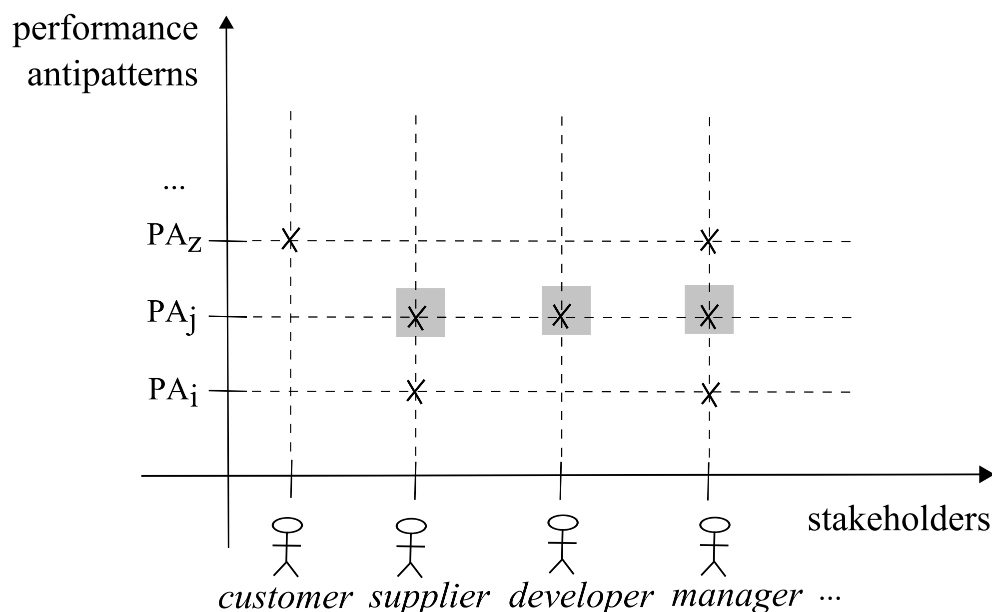


Figure 8.2: Stakeholders analysis as a support for solving antipatterns.

The mapping between performance antipatterns and stakeholders may be useful to give a priority in the sequential solution of antipatterns. For example, in Figure 8.2 we can notice that the antipattern PA_j occurs by considering many stakeholders (i.e. supplier, developer, manager), hence it might be beneficial to start the solution process by applying its refactoring actions, thus to embrace common needs.

In the context of the stakeholders analysis it might be useful to devise methodologies to balance in a proper way the preferences of all the different stakeholders playing a valuable role in the system. In fact different stakeholders might understand and judge with conflicting utilities the multiple performance attributes (e.g. the hardware utilization, the response time of a service).

8.3 Operational profile analysis

The operational profile represents a quantitative characterization of how the software will be used, hence it is essential in any software performance engineering process. More specifically, the operational profile can be expressed as a set of scenarios describing how users interact with the system, i.e. the operations that a software system performs, their probabilities to be invoked, and which dependencies exist in the process [122].

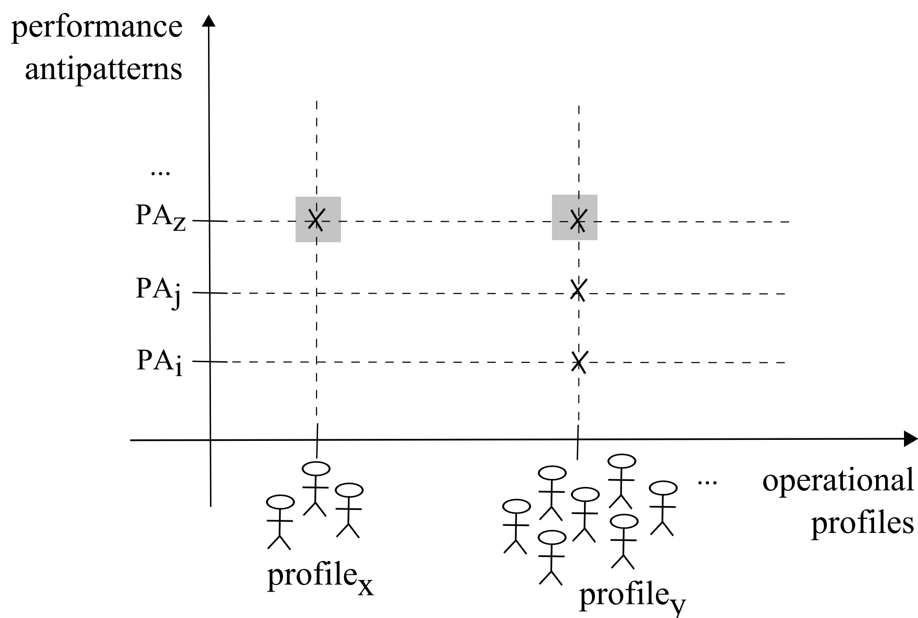


Figure 8.3: Operational profile analysis as a support for solving antipatterns.

Figure 8.3 shows how the operational profile analysis activity can be used to prioritize performance antipatterns: on the x-axis the different operational profiles are reported (e.g. $profile_x$, $profile_y$, $\ldots$); on the y-axis the performance antipatterns instances are listed. A "$\times$" symbol is inserted in the ($profile_x$, PA_i) point if a performance antipattern PA_i occurs when considering the operational profile $profile_x$.

Similarly to the stakeholders analysis, the mapping between performance antipatterns and the operational profile may be useful to give a priority in the sequential solution of antipatterns. For example, in Figure 8.3 we can notice that the antipattern PA_z occurs in both the profiles we consider, hence it might be beneficial to start the solution process by applying its refactoring actions, thus to embrace common needs.

8.4 COST ANALYSIS

The cost analysis activity is meant to predict the costs of the reconfiguration actions aimed at improving the system performance. The evaluation of how much a reconfiguration action costs can be expressed in terms of time necessary to the designer to apply that action, or in terms of the capital outlay. In the following we refer as cost both the amount of effort that software designers need to apply the suggested design alternative and the monetary employment.

Figure 8.4 shows how the cost analysis activity can be used to prioritize performance antipatterns: on the x-axis the cost units are reported; on the y-axis the performance antipatterns instances are listed. A "×" symbol is in the ($cost_x$, PA_i) point by indicating that solving a performance antipattern PA_i implies $cost_x$ units.

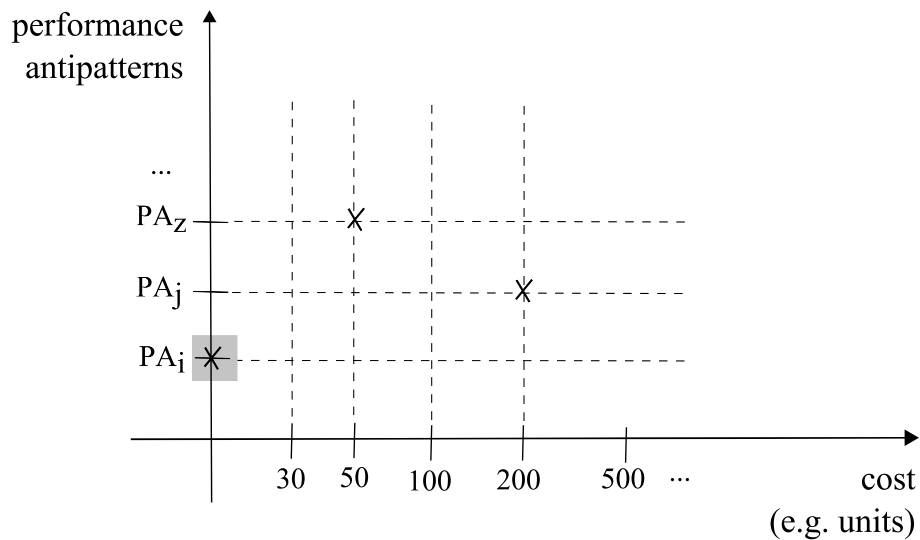


Figure 8.4: Cost analysis as a support for solving antipatterns.

The mapping between performance antipatterns and costs may be useful to give a priority in the sequential solution of antipatterns. For example, in Figure 8.4 we can notice that the PA_j antipattern is the most expensive one. In general hardware solutions are usually more expensive than software solutions, in fact buying a new hardware machine is more expensive than buying a new software component; however, the best strategy is to reuse in a better way the available resources (e.g. re-deploy a software component), without increasing any cost, like for the PA_i antipattern (see Figure 8.4).

Note that several issues might emerge in the process of quantifying the cost of the solutions for some antipatterns. The cost estimation might be restricted only to a subset of the detected antipatterns. For example, the cost of restructuring the database can be more or less expensive, depending on the experience of the software designer that actually performs the operation.

CHAPTER 9

CONCLUSION

In this thesis we dealt with the automated generation of performance feedback in software architectures. We devised a methodology to keep track of the performance knowledge that usually tends to be fragmented and quickly lost, with the purpose of interpreting the performance analysis results and suggesting the most suitable architectural reconfigurations. Such knowledge base is aimed at integrating different forms of data (e.g. architectural model elements, performance indices), in order to support relationships between them and to manage the data over time, while the development advances.

The performance knowledge that we have organized for reasoning on performance analysis results can be considered as an application of data mining to the software performance domain. It has been grouped around design choices and performance model analysis results concepts, thus to act as a data repository available to reason on the performance of a software system. Performance antipatterns have been of crucial relevance in this context since they represent the source of the concepts to identify performance flaws as well as to provide refactorings in terms of architectural alternatives.

The framework we proposed (i.e. PANDA: *P*erformance *A*ntipatterns a*N*d Fee*D*back in Software *A*rchitectures) is aimed at supporting the management of antipatterns. It performs three main activities: *specifying antipatterns*, to define in a well-formed way the properties that lead a software system to reveal a bad practice as well as the changes that provide a solution; *detecting antipatterns*, to locate antipatterns in software architectural models; *solving antipatterns*, to remove the detected performance problems with a set of refactoring actions that can be applied on the software architectural model.

The antipattern-based approach has been validated with two case studies to demonstrate its applicability and validity. We experimented two software modeling notations (i.e. UML and Marte profile, Palladio Component Model) and two performance analysis techniques (i.e. analytic solution, simulation). The validation activity allowed us to identify the benefits of the antipattern-based approach to complex systems in order to assess its suitability to support software designers.

9.1 ACHIEVEMENTS

The aim of this thesis is to achieve a deep insight in the model-based performance analysis of software architectures by interpreting the performance results and generating architectural alternatives able to overcome performance flaws. A list of the main scientific contributions of this thesis is given in the following.

Specifying performance antipatterns. The activity of *specifying antipatterns* has been addressed in [45]: a structured description of the system elements that occur in the definition of antipatterns has been provided, and performance antipatterns have been modeled as logical predicates. Additionally, in [45] the operational counterpart of the antipattern declarative definitions as logical predicates has been implemented with a java rule-engine application. Such engine was able to detect performance antipatterns in an XML representation of the software system that grouped the software architectural model and the performance results data.

A model-driven approach for antipatterns. A Performance Antipattern Modeling Language (PAML), i.e. a metamodel specifically tailored to describe antipatterns, has been introduced in [44]. Such metamodel allows a user-friendly representation of antipatterns, i.e. models expressed by the concepts encoded in PAML. The antipattern representation as PAML-based models allows to manipulate their (neutral) specification. In fact in [44] it has been also discussed a vision on how model-driven techniques (e.g. weaving models [29], difference models [39]) can be used to build a notation-independent approach that addresses the problem of embedding antipatterns knowledge across different modeling notations.

Detecting and solving antipatterns in UML and PCM. The activities of *detecting* and *solving antipatterns* have been currently implemented by defining the antipattern rules and actions into two modeling languages: (i) the UML and Marte profile notation in [42]; (ii) the PCM notation in [129]. In [42] performance antipatterns have been automatically detected in UML models using OCL [107] queries, but we have not yet automated their solution. In [129] a limited set of antipatterns has been automatically detected and solved in PCM models through a benchmark tool. These experiences led us to investigate the expressiveness of UML and PCM modeling languages by classifying the antipatterns in three categories: (i) detectable and solvable; (ii) semi-solvable (i.e. the antipattern solution is only achieved with refactoring actions to be manually performed); (iii) neither detectable nor solvable.

A step ahead in the antipatterns solution. Instead of blindly moving among the antipattern solutions without eventually achieving the desired results, a technique to rank the antipatterns on the basis of their guiltiness for violated requirements has been defined in [47, 46], thus to decide how many antipatterns to solve, which ones and in what order. Experimental results demonstrated the benefits of introducing ranking techniques to support the activity of solving antipatterns.

9.2 Open issues and Future works

Due to the wide range of topics the thesis deals with, there are some open issues in the current version of the framework and many directions can be identified for future works.

9.2.1 Short term goals

In a short term many directions can be identified about future works. A list of the most important ones is given in the following.

Further validation. The approach has to be more extensively validated in order to determine the extent to which it can offer support to user activities. The validation of the approach includes two dimensions: (i) it has to be exposed to a set of target users, such as graduate students in a software engineering course, model-driven developers, more or less experienced software architects, in order to analyze its scope and usability; (ii) it has to be applied to complex case studies by involving industry partners, in order to analyze its scalability. Such experimentation is of worth interest because the final purpose is to integrate the framework in the daily practices of the software development process.

Both the detection and the solution of antipatterns generate some pending issues that give rise to short term goals.

The detection of antipatterns generates the following main categories of open issues:

Accuracy of antipatterns instances. The detection process may introduce false positive/negative instances of antipatterns. We outlined some sources to suitably tune the values of antipatterns boundaries, such as: (i) the system requirements; (ii) the domain expert's knowledge; (iii) the evaluation of the system under analysis. However, threshold values inevitably introduce a degree of uncertainty and extensive experimentation must be done in this direction. Some fuzziness can be introduced for the evaluation of the threshold values [124]. It might be useful to make antipattern detection rules more flexible, and to detect the performance flaws with higher/lower accuracy.

Some metrics are usually used to estimate the efficiency of design patterns detection, such as *precision* (i.e. measuring what fraction of detected pattern occurrences are real) and *recall* (i.e. measuring what fraction of real occurrences are detected). Such metrics do not apply for antipatterns, since the latter ones are not explicitly stated in projects' specifications, due to their nature of capturing bad practices. A confidence value can be associated to an antipattern to quantify the probability that the formula occurrence corresponds to the antipattern presence.

Relationship between antipatterns instances. The detected instances might be related to each other, e.g. one instance can be the generalization or the specialization of another instance. A dependence value can be associated to an antipattern to quantify the probability that its occurrence is dependent from other antipatterns presence.

The solution of antipatterns generates the following main categories of open issues:

No guarantee of performance improvements. The solution of one or more antipatterns does not guarantee performance improvements in advance: the entire process is based on heuristics evaluations. Applying a refactoring action results in a new software architectural model, i.e. a candidate whose performance analysis will reveal if the action has been actually beneficial for the system under study. However, an antipattern-based refactoring action is usually a correctness-preserving transformation that does not alter the semantics of the application, but it may improve the overall performance.

Dependencies of performance requirements. The application of antipattern solutions leads the system to (probably) satisfy the performance requirements covered by such solutions. However, it may happen that a certain number of other requirements get worse. Hence, the new candidate architectural model must take into account at each stage of the process all the requirements, also the previously satisfied ones.

Conflict between antipattern solutions. The solution of a certain number of antipatterns cannot be unambiguously applied due to incoherencies among their solutions. It may happen that the solution of one antipattern suggests to split a component into three finer grain components, while another antipattern at the same time suggests to merge the original component with another one. These two actions obviously contradict each other, although no pre-existing requirement limits their application. Even in cases of no explicit conflict between antipattern solutions, coherency problems can be raised from the order of application of solutions. In fact the result of the sequential application of two (or more) antipattern solutions is not guaranteed to be invariant with respect to the application order. Criteria must be introduced to drive the application order of solutions in these cases. An interesting possibility may be represented by the *critical pairs analysis* [103] that provides a mean to avoid conflicting and divergent refactorings.

9.2.2 Long term goals

In a longer term many directions can be identified about future works. A list of the most important ones is given in the following.

Lack of model parameters. The application of the antipattern-based approach is not limited (in principle) along the software lifecycle, but it is obvious that an early usage is subject to lack of information because the system knowledge improves while the development process progresses. Both the architectural and the performance models may lack of parameters needed to apply the process. For example, internal indices of subsystems that are not yet designed in details cannot be collected. Lack of information, or even uncertainty, about model parameter values can be tackled by analyzing the model piecewise, starting from sub-models, thus to bring insight on the missing parameters.

Influence of domain features. Different cross-cutting concerns such as the workload, the operational profile, etc. usually give rise to different performance analysis results that, in turn, may result in different antipatterns identified in the system. This is a critical issue

and, as usually in performance analysis experiments, the choice of the workload(s) and operational profile(s) must be carefully conducted.

Influence of other software layers. We assume that the performance model only takes into account the (annotated) software architectural model that usually contains information on the software application and hardware platform. Between these two layers there are other components, such as different middlewares and operating systems, that can embed performance antipatterns. The approach shall be extended to these layers for a more accurate analysis of the system. An option can be to integrate benchmarks or models suitable for these layers in our framework.

Limitations from requirements. The application of antipattern solutions can be restricted by functional or non-functional requirements. Example of functional requirements may be legacy components that cannot be split and re-deployed whereas the antipattern solution consists of these actions. Example of non-functional requirements may be budget limitations that do not allow to adopt an antipattern solution due to its extremely high cost. Many other examples can be provided of requirements that (implicitly or explicitly) may affect the antipattern solution activity. For sake of automation such requirements should be pre-defined so that the whole process can take into account them and preventively excluding infeasible solutions.

Consolidate the formalization of performance antipatterns. The Performance Antipatterns Modeling Language (PAML) currently only formalizes the performance problems captured by antipatterns. As future work we plan to complete PAML with a Refactoring Modeling Language (RML) for formalizing the solutions in terms of refactorings, i.e. changes of the original software architectural model.

Note that the formalization of antipatterns reflects our interpretation of the informal literature. Different formalizations of antipatterns can be originated by laying on different interpretations. This unavoidable gap is an open issue in this domain, and certainly requires a wider investigation to consolidate the formal definition of antipatterns. Logical predicates of antipatterns can be further refined by looking at probabilistic model checking techniques, as experimented in [67].

Investigate architectural description languages. The framework is currently considering two modeling notations: UML and PCM. In general, the subset of target modeling languages can be enlarged as far as the concepts for representing antipatterns are available; for example, architectural description languages such as AADL [10] can be also suited to validate the approach. A first investigation has been already conducted on how to specify, detect, and solve performance antipatterns in the Æmilia architectural language [26], however it still requires a deep experimentation.

Multi-objective goals. The framework currently considers only the performance goals of software systems. It can be extended to other quantitative quality criteria of software architectures such as reliability, security, etc., thus to support trade-off decisions between multiple quality criteria.

REFERENCES

[1] Compuware. Applied performance management survey (October 2006), *http://www.cnetdirectintl.com/direct/compuware/Ovum_APM/APM_Survey_Report.pdf*.

[2] E. Friedman-Hill, Jess, the Rule Engine for the Java Platform, *http://herzberg.ca.sandia.gov/jess/*.

[3] EMF project web site, *http://www.eclipse.org/modeling/emf/*.

[4] Extensible Markup Language (XML), W3C, *http://www.w3.org/XML*.

[5] http://www.di.univaq.it/catia.trubiani/phdthesis/ECS.xml.

[6] http://www.di.univaq.it/catia.trubiani/phdthesis/PA_xmlschema.xsd.

[7] Object Management Group (OMG), *http://www.omg.org*.

[8] Package org.eclipse.emf.ecore, *http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/package-summary.html*.

[9] Package org.w3c.dom, *http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html*.

[10] SAE, Architecture Analysis and Design Language (AADL), June 2006, as5506/1, http://www.sae.org.

[11] Series Z: Languages And General Software Aspects For Telecommunication Systems, *http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf*.

[12] UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005), *http://www.omg.org/cgi-bin/doc?formal/05-07-04*.

[13] UML Profile for MARTE, OMG, *http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf*.

[14] UML Profile for Schedulability, Performance, and Time Specification, SPT, OMG document formal/03-09-01, Object Management Group, Inc. (2003), *http://www.omg.org/cgi-bin/doc?formal/03-09-01*.

[15] ADAMSON, A., BONDI, A. B., JUIZ, C., AND SQUILLANTE, M. S., Eds. *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010* (2010), ACM.

[16] ALETI, A., BJÖRNANDER, S., GRUNSKE, L., AND MEEDENIYA, I. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *MOMPES* (2009), pp. 61–71.

[17] AMYOT, D., LOGRIPPO, L., BUHR, R. J. A., AND GRAY, T. Use case maps for the capture and validation of distributed systems requirements. In *RE* (1999), IEEE Computer Society, pp. 44–.

[18] A.W.ROSCOE. *Theory and Practice of Concurrency.* Prentice Hall, 1997.

[19] BALSAMO, S., DI MARCO, A., INVERARDI, P., AND SIMEONI, M. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Software Eng. 30*, 5 (2004), 295–310.

[20] BANKS, J., CARSON, J. S., NELSON, B. L., AND NICOL, D. M. *Discrete-Event System Simulation (3rd Edition).* Prentice Hall, 2000.

[21] BARBER, K. S., GRASER, T. J., AND HOLT, J. Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation. In *ASE* (2002), pp. 172–182.

[22] BECKER, S., KOZIOLEK, H., AND REUSSNER, R. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software 82*, 1 (2009), 3–22.

[23] BERGER, J. O. *Statistical Decision Theory and Bayesian Analysis*. Springer, 1985.

[24] BERNARDO, M., AND BRAVETTI, M. Performance measure sensitive congruences for markovian process algebras. *Theor. Comput. Sci. 290*, 1 (2003), 117–160.

[25] BERNARDO, M., CIANCARINI, P., AND DONATIELLO, L. On the formalization of architectural types with process algebras. In *SIGSOFT FSE* (2000), pp. 140–148.

[26] BERNARDO, M., DONATIELLO, L., AND CIANCARINI, P. Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In *Performance* (2002), M. Calzarossa and S. Tucci, Eds., vol. 2459 of *Lecture Notes in Computer Science*, Springer, pp. 236–260.

[27] BERNARDO, M., AND GORRIERI, R. A tutorial on empa: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theor. Comput. Sci. 202*, 1-2 (1998), 1–54.

[28] BERNARDO, M., AND GORRIERI, R. Corrigendum to "a tutorial on empa: a theory of concurrent processes with nondeterminism, priorities, probabilities and time" - [tcs 202 (1998) 1-54]. *Theor. Comput. Sci. 254*, 1-2 (2001), 691–694.

[29] BÉZIVIN, J. On the unification power of models. *Software and System Modeling 4*, 2 (2005), 171–188.

[30] BONDAREV, E., CHAUDRON, M. R. V., AND DE KOCK, E. A. Exploring performance trade-offs of a JPEG decoder using the deepcompass framework. In *WOSP* (2007), pp. 153–163.

[31] BORODAY, S., PETRENKO, A., SINGH, J., AND HALLAL, H. Dynamic Analysis of Java Applications for MultiThreaded Antipatterns. In *Workshop on Dynamic Analysis (WODA)* (2005), pp. 1–7.

[32] BROOKNER, E. *Tracking and Kalman Filtering Made Easy*. Wiley Interscience, 1998.

[33] BROWN, W. J., MALVEAU, R. C., III, H. W. M., AND MOWBRAY, T. J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1998.

[34] BUHR, R. J. A. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Trans. Software Eng. 24*, 12 (1998), 1131–1155.

[35] CANFORA, G., PENTA, M. D., ESPOSITO, R., AND VILLANI, M. L. An approach for QoS-aware service composition based on genetic algorithms. In *GECCO* (2005), H.-G. Beyer and U.-M. O'Reilly, Eds., ACM, pp. 1069–1075.

[36] CARDELLI, L., AND GORDON, A. D. Mobile ambients. In *FoSSaCS* (1998), M. Nivat, Ed., vol. 1378 of *Lecture Notes in Computer Science*, Springer, pp. 140–155.

[37] CHEN, K., SZTIPANOVITS, J., ABDELWAHED, S., AND JACKSON, E. K. Semantic Anchoring with Model Transformations. In *ECMDA-FA* (2005), pp. 115–129.

[38] CHIOLA, G., MARSAN, M. A., BALBO, G., AND CONTE, G. Generalized stochastic petri nets: A definition at the net level and its implications. *IEEE Trans. Software Eng. 19*, 2 (1993), 89–107.

[39] CICCHETTI, A., RUSCIO, D. D., AND PIERANTONIO, A. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology 6*, 9 (2007), 165–185.

[40] CLEMENTS, P., BACHMANN, F., BASS, L., GARLAN, D., IVERS, J., LITTLE, R., NORD, R., AND STAFFORD, J. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003.

[41] CLEMENTS, P. C., GARLAN, D., LITTLE, R., NORD, R. L., AND STAFFORD, J. A. Documenting software architectures: Views and beyond. In *ICSE* (2003), IEEE Computer Society, pp. 740–741.

[42] CORTELLESSA, V., DI MARCO, A., ERAMO, R., PIERANTONIO, A., AND TRUBIANI, C. Digging into UML models to remove performance antipatterns. In *ICSE Workshop Quovadis* (2010), pp. 9–16.

[43] CORTELLESSA, V., AND FRITTELLA, L. A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis. In *Proceedings of the Formal Methods and Stochastic Models for Performance Evaluation, Fourth European Performance Engineering Workshop, EPEW 2007* (2007), pp. 171–185.

[44] CORTELLESSA, V., MARCO, A. D., ERAMO, R., PIERANTONIO, A., AND TRUBIANI, C. Approaching the Model-Driven Generation of Feedback to Remove Software Performance Flaws. In *EUROMICRO-SEAA* (2009), IEEE Computer Society, pp. 162–169.

[45] CORTELLESSA, V., MARCO, A. D., AND TRUBIANI, C. Performance Antipatterns as Logical Predicates. In *ICECCS* (2010), R. Calinescu, R. F. Paige, and M. Z. Kwiatkowska, Eds., IEEE Computer Society, pp. 146–156.

[46] CORTELLESSA, V., MARTENS, A., REUSSNER, R., AND TRUBIANI, C. A Process to Effectively Identify "Guilty" Performance Antipatterns. In Rosenblum and Taentzer [114], pp. 368–382.

[47] CORTELLESSA, V., MARTENS, A., REUSSNER, R., AND TRUBIANI, C. Towards the identification of "Guilty" performance antipatterns. In Adamson et al. [15], pp. 245–246.

[48] CORTELLESSA, V., AND MIRANDOLA, R. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Sci. Comput. Program. 44*, 1 (2002), 101–129.

[49] DAMM, W., AND HAREL, D. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design 19*, 1 (2001), 45–80.

[50] DI MARCO, A. ADL aspects of AEmilia, 2008. Lecture class notes.

[51] DOBRZANSKI, L., AND KUZNIARZ, L. An approach to refactoring of executable UML models. In *ACM Symposium on Applied Computing (SAC)* (2006), pp. 1273–1279.

[52] DUDNEY, B., ASBURY, S., KROZAK, J. K., AND WITTKOPF, K. *J2EE Antipatterns*. 2003.

[53] EGOR BONDAREV, MICHEL R.V. CHAUDRON, J. Z., AND KLOMP, A. Quality-Oriented Design Space Exploration for Component-Based Architectures. Tech. rep., Eindhoven University of Technology and LogicaCMG Nederland, The Netherlands, 2006.

[54] EHRGOTT, M. *Multicriteria Optimization*. 2005.

[55] ELAASAR, M., BRIAND, L. C., AND LABICHE, Y. A Metamodeling Approach to Pattern Specification. In *MoDELS* (2006), O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199 of *Lecture Notes in Computer Science*, Springer, pp. 484–498.

[56] EPIFANI, I., GHEZZI, C., MIRANDOLA, R., AND TAMBURRELLI, G. Model evolution by run-time parameter adaptation. In *ICSE* (2009), IEEE, pp. 111–121.

[57] FEILER, P. H., GLUCH, D. P., AND HUDAK, J. J. The Architecture Analysis and Design Language (AADL): An Introduction. Tech. Rep. CMU/SEI-2006-TN-001, Software Engineering Institute, Carnegie Mellon University, 2006.

[58] FIELD, T., HARRISON, P. G., BRADLEY, J. T., AND HARDER, U., Eds. *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings* (2002), vol. 2324 of *Lecture Notes in Computer Science*, Springer.

[59] FRANCE, R. B., KIM, D.-K., GHOSH, S., AND SONG, E. A uml-based pattern specification technique. *IEEE Trans. Software Eng. 30*, 3 (2004), 193–206.

[60] FRANKS, G., HUBBARD, A., MAJUMDAR, S., NEILSON, J. E., PETRIU, D. C., ROLIA, J. A., AND WOODSIDE, C. M. A toolset for performance engineering and software design of client-server systems. *Perform. Eval. 24*, 1-2 (1995), 117–136.

[61] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addision-Wesley, 1995.

[62] GARLAN, D. Research directions on software architecture. *ACM Comput. Surv. 27*, 2 (1995), 257–261.

[63] GARLAN, D., AND PERRY, D. E. Software architecture: Practice, potential, and pitfalls. In *ICSE* (1994), pp. 363–364.

[64] GARLAN, D., AND SHAW, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering* (Singapore, 1993), V. Ambriola and G. Tortora, Eds., World Scientific Publishing Company, pp. 1–39.

[65] GILMORE, S., AND HILLSTON, J. The pepa workbench: A tool to support a process algebra-based approach to performance modelling. In *Computer Performance Evaluation* (1994), G. Haring and G. Kotsis, Eds., vol. 794 of *Lecture Notes in Computer Science*, Springer, pp. 353–368.

[66] GIRAULT, C., AND VALK, R. *Petri Nets for Systems Engineering*. Springer, 2002.

[67] GRUNSKE, L. Specification patterns for probabilistic quality properties. In *ICSE* (2008), W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, pp. 31–40.

[68] GU, G. P., AND PETRIU, D. C. XSLT transformation from UML models to LQN performance models. In *Workshop on Software and Performance* (2002), pp. 227–234.

[69] GU, G. P., AND PETRIU, D. C. From UML to LQN by XML algebra-based model transformations. In *Workshop on Software and Performance, WOSP* (2005), pp. 99–110.

[70] HARMAN, M. The Current State and Future of Search Based Software Engineering. In *FOSE* (2007), pp. 342–357.

[71] HARMAN, M. The relationship between search based software engineering and predictive modeling. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (New York, NY, USA, 2010), ACM, pp. 1:1–1:13.

[72] HARMAN, M. Why the Virtual Nature of Software Makes It Ideal for Search Based Optimization. In Rosenblum and Taentzer [114], pp. 1–12.

[73] HARMAN, M., MANSOURI, S. A., AND ZHANG, Y. Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Tech. Rep. TR-09-03, 2009.

[74] HARRELD, H. NASA Delays Satellite Launch After Finding Bugs in Software Program, April 20, 1998. Federal Computer Week.

[75] HARRELD, H. Panel Slams EOSDIS, September 14, 1998. Federal Computer Week.

[76] HAUCK, M., KUPERBERG, M., KROGMANN, K., AND REUSSNER, R. Modelling Layered Component Execution Environments for Performance Prediction. In *CBSE* (2009), G. A. Lewis, I. Poernomo, and C. Hofmeister, Eds., vol. 5582 of *Lecture Notes in Computer Science*, Springer, pp. 191–208.

[77] HERMANNS, H., HERZOG, U., AND KATOEN, J.-P. Process algebra for performance evaluation. *Theor. Comput. Sci. 274*, 1-2 (2002), 43–87.

[78] HERMANNS, H., HERZOG, U., KLEHMET, U., MERTSIOTAKIS, V., AND SIEGLE, M. Compositional performance modelling with the TIPPtool. *Perform. Eval. 39*, 1-4 (2000), 5–35.

[79] IPEK, E., MCKEE, S. A., CARUANA, R., DE SUPINSKI, B. R., AND SCHULZ, M. Efficiently exploring architectural design spaces via predictive modeling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 195–206.

[80] IPEK, E., MCKEE, S. A., SINGH, K., CARUANA, R., DE SUPINSKI, B. R., AND SCHULZ, M. Efficient architectural design space exploration via predictive modeling. *ACM Transactions on Architecture and Code Optimization (TACO) 4*, 4 (2008).

[81] JAIN, R. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. *SIGMETRICS Performance Evaluation Review 19*, 2 (1991), 5–11.

[82] JOHN E. HOPCROFT, RAJEEV MOTWANI, J. D. U. *Introduction to Automata Theory, Languages, and Computation*, 2 ed. Addison Wesley, 2000.

[83] KANT, K. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, 1992.

[84] KAPOVÁ, L., BUHNOVA, B., MARTENS, A., HAPPE, J., AND REUSSNER, R. State dependence in performance evaluation of component-based software systems. In Adamson et al. [15], pp. 37–48.

[85] KAVIMANDAN, A., AND GOKHALE, A. S. Applying Model Transformations to Optimizing Real-Time QoS Configurations in DRE Systems. In *QoSA* (2009), R. Mirandola, I. Gorton, and C. Hofmeister, Eds., vol. 5581 of *Lecture Notes in Computer Science*, Springer, pp. 18–35.

[86] KELLER, R. M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (1976), 371–384.

[87] KEMENY, J., AND SNELL, J. *Finite Markov Chains*. Springer, 1976.

[88] KLEINROCK, L. *Queueing Systems Vol. 1:Theory*. Wiley, 1975.

[89] KOZIOLEK, H. Performance evaluation of component-based software systems: A survey. *Perform. Eval. 67*, 8 (2010), 634–658.

[90] KOZIOLEK, H., AND REUSSNER, R. A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In *SIPEW* (2008), S. Kounev, I. Gorton, and K. Sachs, Eds., vol. 5119 of *Lecture Notes in Computer Science*, Springer, pp. 58–78.

[91] KROGMANN, K., KUPERBERG, M., AND REUSSNER, R. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *IEEE Transactions on Software Engineering* (2010). accepted for publication, to appear.

[92] LAZOWSKA, E., KAHORJAN, J., GRAHAM, G. S., AND SEVCIK, K. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[93] LIN, Y., ZHANG, J., AND GRAY, J. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development* (2004).

[94] MALAVOLTA, I., MUCCINI, H., PELLICCIONE, P., AND TAMBURRI, D. A. Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. *IEEE Trans. Software Eng. 36*, 1 (2010), 119–140.

[95] MARKOV, A. A. *Extension of the limit theorems of probability theory to a sum of variables connected in a chain*. John Wiley and Sons, 1971.

[96] MARSAN, M. A., BALBO, G., CHIOLA, G., AND DONATELLI, S. On the product-form solution of a class of multiple-bus multiprocessor system models. *Journal of Systems and Software 6*, 1-2 (1986), 117–124.

[97] MARSAN, M. A., CONTE, G., AND BALBO, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst. 2*, 2 (1984), 93–122.

[98] MARTENS, A., AND KOZIOLEK, H. Performance-oriented Design Space Exploration. In *Workshop on Component-Oriented Programming (WCOP'08), Karlsruhe, Germany* (2008), pp. 25–32.

[99] MARTENS, A., KOZIOLEK, H., BECKER, S., AND REUSSNER, R. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In Adamson et al. [15], pp. 105–116.

[100] MCCARTHY, J., ABRAHAMS, P., EDWARDS, D. J., AND T. P. HART, I. L. *LISP Programmer's Manual*. MIT Press, 1985.

[101] MCGREGOR, J. D., BACHMANN, F., BASS, L., BIANCO, P., , AND KLEIN, M. Using arche in the classroom: One experience. Tech. Rep. CMU/SEI-2007-TN-001, Software Engineering Institute, Carnegie Mellon University, 2007.

[102] MEDVIDOVIC, N., AND TAYLOR, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering (TSE) 26*, 1 (2000), 70–93.

[103] MENS, T., TAENTZER, G., AND RUNGE, O. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electr. Notes Theor. Comput. Sci. 127*, 3 (2005), 113–128.

[104] MILNER, R. *Communication and Concurrency*. Prentice Hall, 1989.

[105] MILNER, R. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[106] OBJECT MANAGEMENT GROUP (OMG). Lightweight CCM RFP, 2002. OMG Document realtime/02-11-27.

[107] OBJECT MANAGEMENT GROUP (OMG). OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.

[108] PARSONS, T., AND MURPHY, J. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology 7*, 3 (2008), 55–91.

[109] PERRY, D. E., AND WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes 17* (October 1992), 40–52.

[110] PETRIU, D. C., AND SHEN, H. Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In Field et al. [58], pp. 159–177.

[111] PETRIU, D. C., AND WOODSIDE, C. M. Software performance models from system scenarios in use case maps. In Field et al. [58], pp. 141–158.

[112] REISIG, W. *Petri Nets: An Introduction*, vol. 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[113] RIVERA, J. E., AND VALLECILLO, A. Representing and Operating with Model Differences. In *TOOLS (46)* (2008), R. F. Paige and B. Meyer, Eds., vol. 11 of *Lecture Notes in Business Information Processing*, Springer, pp. 141–160.

[114] ROSENBLUM, D. S., AND TAENTZER, G., Eds. *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (2010), vol. 6013 of *Lecture Notes in Computer Science*, Springer.

[115] SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. *IEEE Computer 39*, 2 (2006), 25–31.

[116] SMITH, C. U. Introduction to software performance engineering: Origins and outstanding problems. In *SFM* (2007), M. Bernardo and J. Hillston, Eds., vol. 4486 of *Lecture Notes in Computer Science*, Springer, pp. 395–428.

[117] SMITH, C. U., AND MILLSAP, C. V. Software performance engineering for oracle applications: Measurements and models. In *Int. CMG Conference* (2008), Computer Measurement Group, pp. 331–342.

[118] SMITH, C. U., AND WILLIAMS, L. G. Performance engineering evaluation of corba-based distributed systems with spe*ed. In *Computer Performance Evaluation (Tools)* (1998), R. Puigjaner, N. N. Savino, and B. Serra, Eds., vol. 1469 of *Lecture Notes in Computer Science*, Springer, pp. 321–335.

[119] SMITH, C. U., AND WILLIAMS, L. G. Software performance antipatterns. In *Workshop on Software and Performance* (2000), pp. 127–136.

[120] SMITH, C. U., AND WILLIAMS, L. G. Software Performance AntiPatterns; Common Performance Problems and their Solutions. In *International Computer Measurement Group Conference* (2001), pp. 797–806.

[121] SMITH, C. U., AND WILLIAMS, L. G. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *International Computer Measurement Group Conference* (2002), pp. 667–674.

[122] SMITH, C. U., AND WILLIAMS, L. G. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. 2002.

[123] SMITH, C. U., AND WILLIAMS, L. G. More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *International Computer Measurement Group Conference* (2003), pp. 717–725.

[124] SO, S. S., CHA, S. D., AND KWON, Y. R. Empirical evaluation of a fuzzy logic-based software quality prediction model. *Fuzzy Sets and Systems 127*, 2 (2002), 199–208.

[125] SOMMERVILLE, I. *Software Engineering*, 8 ed. Addison Wesley, 2006.

[126] STEIN, D., HANENBERG, S., AND UNLAND, R. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In *MDAFA* (2004), U. Aßmann, M. Aksit, and A. Rensink, Eds., vol. 3599 of *Lecture Notes in Computer Science*, Springer, pp. 77–92.

[127] TATE, B., CLARK, M., LEE, B., AND LINSKEY, P. *Bitter EJB*. 2003.

[128] TRCKA, N., VAN DER AALST, W. M., AND SIDOROVA, N. Data-Flow Anti-Patterns: Discovering Dataflow Errors in Workflows. In *Conference on Advanced Information Systems (CAiSE)* (2009), vol. 5565, LNCS Springer, pp. 425–439.

[129] TRUBIANI, C., AND KOZIOLEK, A. Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models. accepted to International Conference on Performance Engineering (ICPE) 2011, to appear.

[130] WILLIAMS, L. G., AND SMITH, C. U. PASA(SM): An Architectural Approach to Fixing Software Performance Problems. In *International Computer Measurement Group Conference* (2002), pp. 307–320.

[131] WILLIAMS, L. G., AND SMITH, C. U. Software performance engineering: A tutorial introduction. In *Int. CMG Conference* (2007), Computer Measurement Group, pp. 387–398.

[132] WOODSIDE, C. M. A Three-View Model for Performance Engineering of Concurrent Software. *IEEE Transactions on Software Engineering (TSE) 21*, 9 (1995), 754–767.

[133] WOODSIDE, C. M., FRANKS, G., AND PETRIU, D. C. The Future of Software Performance Engineering. In *FOSE* (2007), pp. 171–187.

[134] WOODSIDE, C. M., HRISCHUK, C. E., SELIC, B., AND BAYAROV, S. Automated performance modeling of software generated by a design environment. *Perform. Eval. 45*, 2-3 (2001), 107–123.

[135] WOODSIDE, C. M., MONTEALEGRE, J. R., AND BUHR, R. J. A. A performance model for hardware/software issues in computer-aided design of protocol systems. *Computer Communication Review 14*, 2 (1984), 132–139.

[136] WOODSIDE, C. M., NEILSON, J. E., PETRIU, D. C., AND MAJUMDAR, S. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Computers 44*, 1 (1995), 20–34.

[137] WOODSIDE, C. M., PETRIU, D. C., PETRIU, D. B., SHEN, H., ISRAR, T., AND MERSEGUER, J. Performance by unified model analysis (PUMA). In *WOSP* (2005), pp. 1–12.

[138] XU, J. Rule-based automatic software performance diagnosis and improvement. *Perform. Eval. 67*, 8 (2010), 585–611.

[139] ZHENG, T., LITOIU, M., AND WOODSIDE, C. M. Integrated Estimation and Tracking of Performance Model Parameters with Autoregressive Trends. accepted to International Conference on Performance Engineering (ICPE) 2011, to appear.

[140] ZHENG, T., AND WOODSIDE, C. M. Heuristic optimization of scheduling and allocation for distributed systems with soft deadlines. In *Computer Performance Evaluation / TOOLS* (2003), P. Kemper and W. H. Sanders, Eds., vol. 2794 of *Lecture Notes in Computer Science*, Springer, pp. 169–181.

[141] ZHENG, T., WOODSIDE, C. M., AND LITOIU, M. Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng. 34*, 3 (2008), 391–406.

APPENDIX A

AN XML SCHEMA FOR PERFORMANCE ANTIPATTERN ELEMENTS

In this Appendix we provide a structured description of the architectural model elements that occur in the definitions of antipatterns [123] (such as software entity, hardware utilization, operation throughput, etc.), which is meant to be the basis for a definition of antipatterns as logical predicates (see Section 3.4).

Since an (annotated) software architectural model contains details that are not relevant for the antipattern definition, as a first step we have filtered the minimal amount of concepts we need for the antipatterns definition. These concepts have been organized in an XML Schema, i.e. aimed at detecting performance issues, not representing software systems. The choice of XML [4] as representation language comes from its primary nature of interchange format supported by many tools that makes it one of the most straightforward means to define generic structured data.

Software architectural model elements are organized in views. We consider three different views representing three sources of information: the *Static View* that captures the modules (e.g. classes, components, etc.) involved in the software system and the relationships among them; the *Dynamic View* that represents the interactions that occur between the modules to provide the system functionalities; and finally the *Deployment View* that describes the mapping of the modules onto platform sites. This organization stems from the Three-View Model that was introduced in [132] for performance engineering of software systems.

Overlaps among views can occur. For example, the elements interacting in the dynamics of a system are also part of the static and the deployment views. In particular, we adopt the term *Service* to represent the high-level functionalities of the software system that are meant to include all the interacting elements among the three views. To avoid redundancy and consistency problems, concepts shared by multiple views are defined once in a view, and simply referred in the other ones (through XML RFID).

The XML Schema we propose for performance antipatterns [6] is synthetically shown in Figure A.1: a *System* has an identifier (*systemID*) and it is composed of a set of *ModelElements* belonging to the three different Views, and of the set of *Functionalities* it

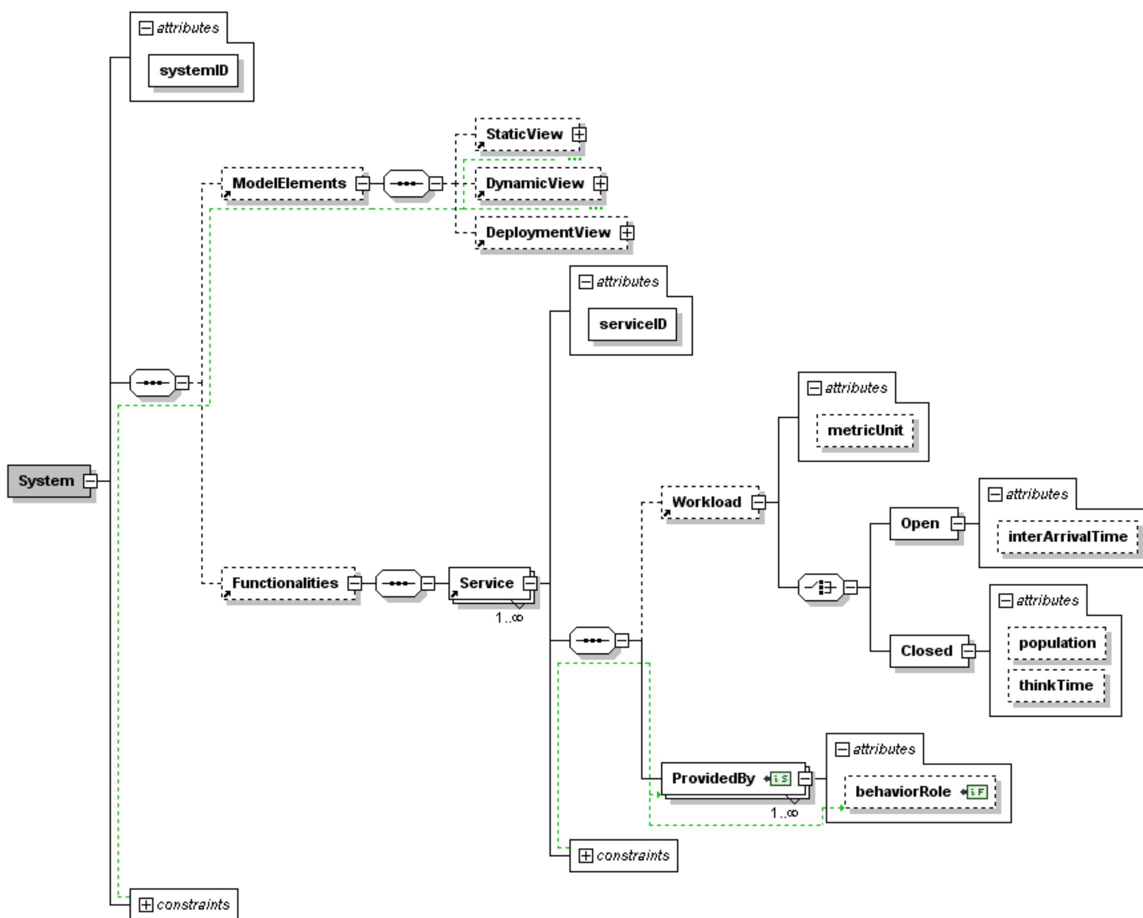


Figure A.1: An excerpt of the XML Schema.

provides. The *Static* View groups the elements needed to specify structural aspects of the software system; the *Dynamic* View deals with the behavior of the system; and finally the *Deployment* View captures the elements of the deployment configuration.

A *Service* has an identifier (*serviceID*) and it can be associated to a *Workload*, i.e. *Open* (specified by the *interArrivalTime*, e.g. a new request arrives each 0.5 seconds) or *Closed* (specified by the *population* and the *thinkTime*, e.g. there is a population of 25 requests entering the system, executing their operational profile, and then re-entering the system after a think time of 2 minutes). Note that the measurement unit (e.g. micro seconds, seconds, minutes) for the workload is specified in the *metricUnit* attribute. In fact the *metricUnit* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *msec* (referring to micro seconds), *sec* (referring to seconds), *min* (referring to minutes), and *other* (referring to any other metric unit customized by the user).

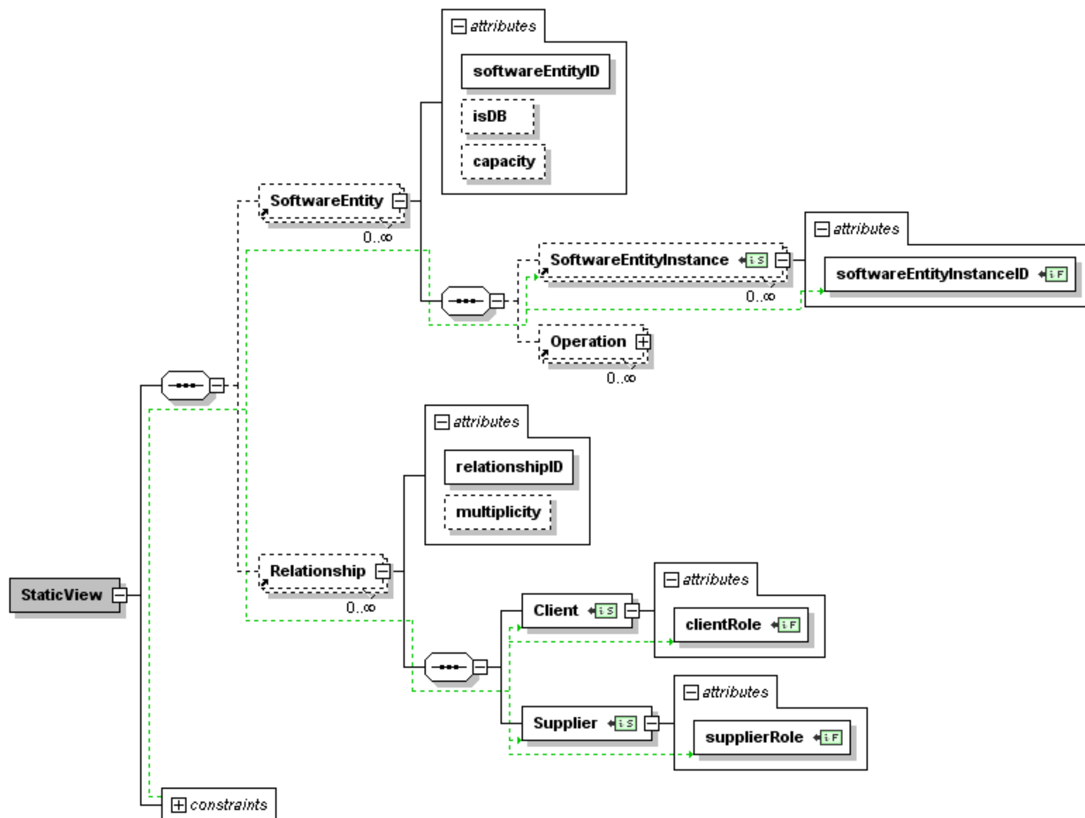
Each service contains static, dynamic and deployment elements; the reference to the architectural model elements is obtained by specifying that a service is *ProvidedBy* a set of *Behavior*s whose id reference (the behaviorID attribute, see Figure A.6(a)) is referred in the attribute *behaviorRole*. It will be the behavior (belonging to the Dynamic View) to contain all the other references among the model elements belonging to the other views (i.e. Static and Deployment).

A.1 STATIC VIEW

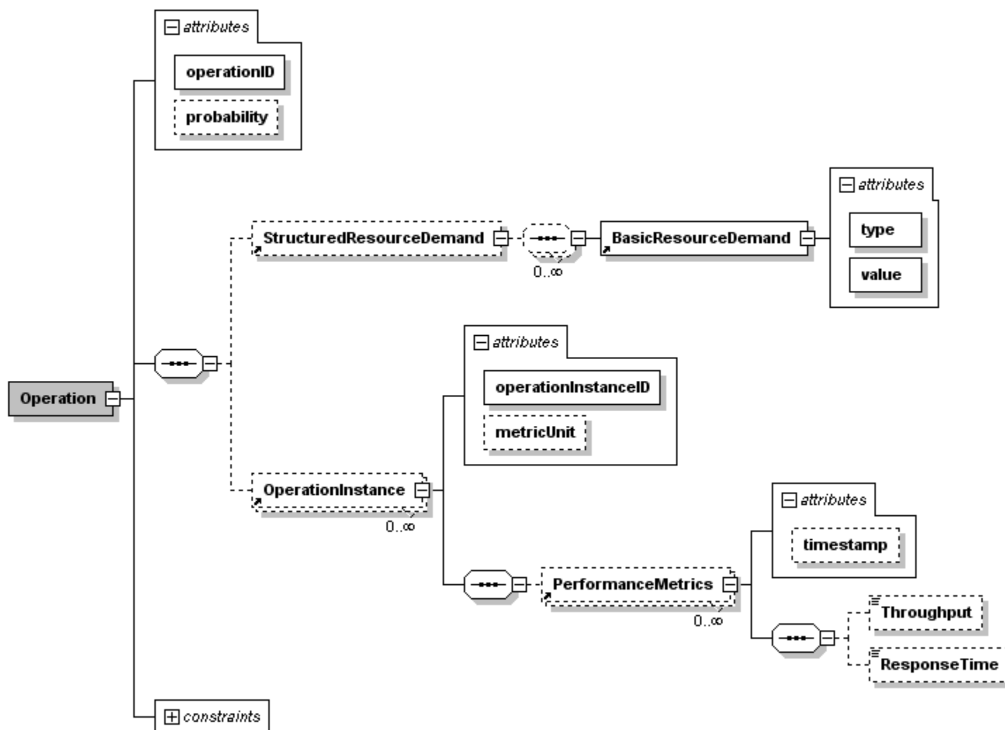
The *Static View* contains elements to describe the static aspects of the system, it is composed of a set of *SoftwareEntity* and *Relationship* model elements.

The *SoftwareEntity* element has an identifier (*softwareEntityID*), a boolean value to specify if it is a database (*isDB*), an integer value to specify its pool size (*capacity*). A software entity contains a set of *SoftwareEntityInstance* elements specified by their identifiers (*softwareEntityInstanceID*), and a set of *Operation*(s).

A *Relationship* has an identifier (*relationshipID*), its *multiplicity*, and it contains a *Client* and a *Supplier*. Both these elements have an attribute, i.e. *clientRole* and *supplierRole* respectively, that refers to a software entity instance identifier (*softwareEntityInstanceID*) previously declared. From a performance perspective, the only interesting relationship between two software entities is the usage relationship. Thus, the XML Schema does not contains elements for all relationships between two software entities (e.g. association, aggregation, etc.), but it flattens any of them in a client/supplier one. For example, in the aggregation relationship, a software entity aggregates one or more instances of another software entity in order to use their methods. In our Schema this is represented as a *Relationship* where the first software entity instance has a *clientRole* and the aggregated ones has a *supplierRole*. In fact, we recall that the XML Schema we propose is not meant to represent software systems, but only to organize all the concepts providing useful information for the performance antipatterns.



(a) *SoftwareEntity* and *Relationship* elements.



(b) *Operation* element.

Figure A.2: XML Schema - *Static View*.

The details of the *Operation* element are shown in Figure A.2(b). An *Operation* has an identifier (*operationID*), and the *probability* of its occurrence. Besides, an operation has a *StructuredResourceDemand* composed by multiple *BasicResourceDemand*(s). A basic resource demand is composed by the resource *type* (e.g. cpu work units, database accesses, and network messages), and its *value* (e.g. number of cpu instructions, number of DB accesses, and number of messages sent over the network). Note that the resource *type* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *computation* (referring to cpu work units), *storage* (referring to database accesses), *bandwidth* (referring to network messages), and *other* (referring to any other resource type customized by the user).

An operation contains a set of *OperationInstance*s specified with their identifiers (*operationInstanceID*), and some performance metrics are associated to each instance, namely *PerformanceMetrics*. The *metricUnit* attribute denotes the measurement unit adopted (e.g. micro seconds, seconds, minutes), and they can be either *Throughput* or *ResponseTime*. Multiple values can be specified for these two latter indices, and they represent the results from the simulation or the monitoring of the system over time. In fact they can be evaluated at a certain date and/or time (*timestamp*), thus to capture the multiple-values antipatterns.

Figures A.3, A.4, A.5 show some examples of elements belonging to the Static View of the XML Schema: the left sides of these Figures give graphical representations of a software architectural model, whereas the right sides report excerpts of the XML files (compliant to the XML Schema) representing the features of the software architectural model.

Figure A.3 shows an example of the *Relationship* element. There are two software entity instances, i.e. the webServer $w1$ with a pool size of ten requests, and the database $d1$, and they are related each other through many connections. The *Relationship* involves the webServer in the *clientRole* and the database in the *supplierRole*; the number of operations required from the client to the supplier (i.e. getUserName, getUserAddress, getUserPassport, getUserWorkInfo), that is four, is stored in the *multiplicity* attribute of the *Relationship* element.

Figure A.4 shows an example of the *StructuredResourceDemand* element. There is a Service (authorizeTransaction), that is provided by means of three operations (i.e. validateUser, validateTransaction, sendResult). The operation validateUser requires the following amount of resource demand: 1 *computation* unit, 2 *storage* units, and 0 *bandwidth* unit. We recall that computation, storage, and bandwidth represent the enumeration values for the basic resource demand *type* element of the XML Schema. The *Operation* validateUser that has two *OperationInstance*s (i.e. v1 and v2) that will be referred in Figure A.5.

Figure A.5 shows an example of the *PerformanceMetrics* element. The *OperationInstance*s v1 and v2 are deployed on different processing nodes, i.e. proc1 and proc2. Note that performance metrics can be obtained by simulating or monitoring the performance

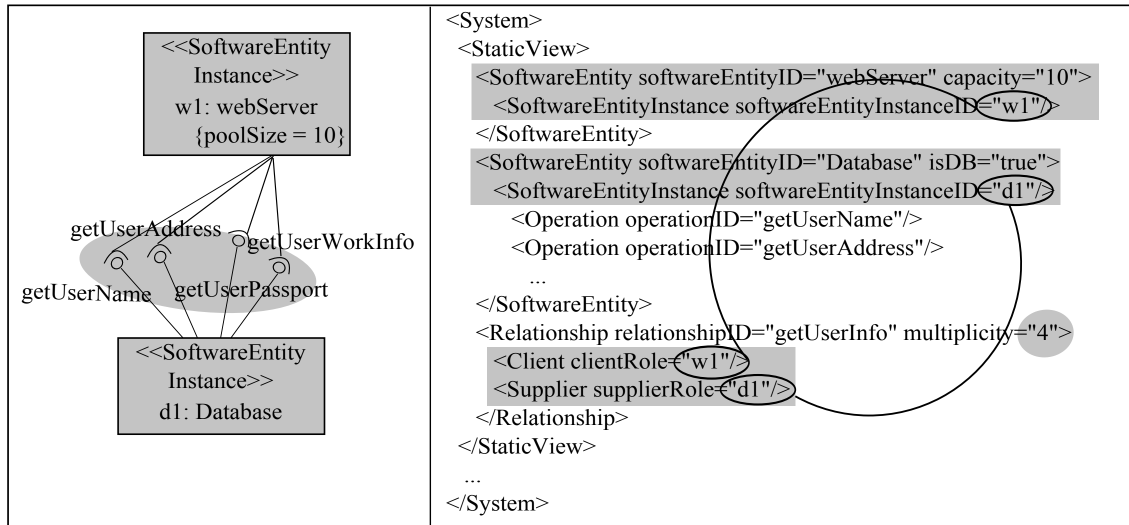


Figure A.3: An example of the *Relationship* element.

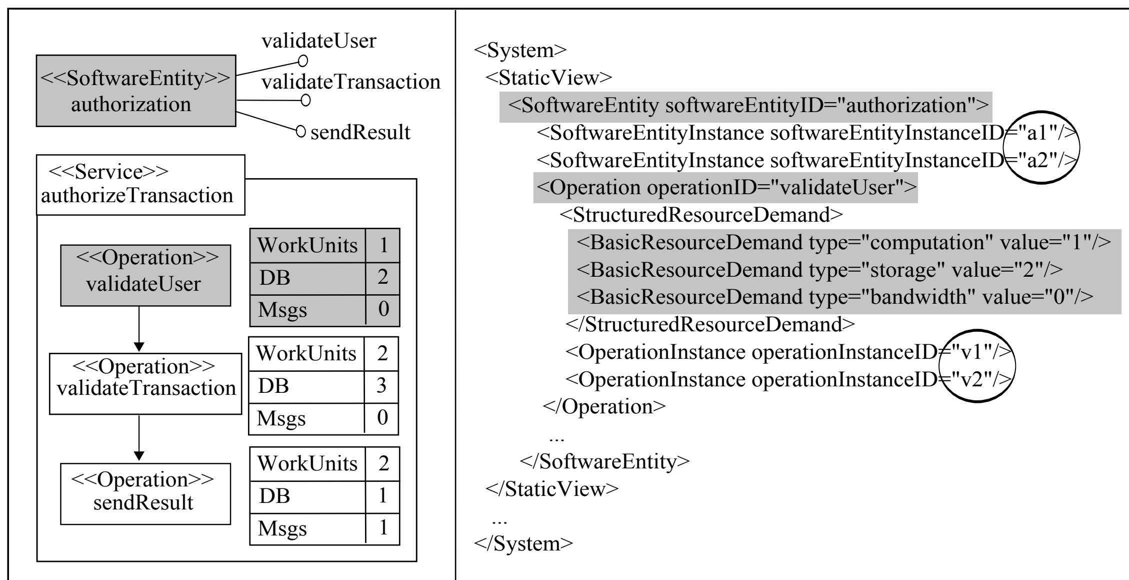


Figure A.4: An example of the *StructuredResourceDemand* element.

model or by solving it analytic or other ways. In the example that we propose in Figure A.5 both solutions are applied: (i) the *OperationInstance* v1 is evaluated by simulating the performance model, and the simulation reveals that after the system is running for 10 seconds there are the following performance metrics: *Throughput* = 100 requests/second, *ResponseTime* = 1.8 seconds; whereas after the system is running for 100 seconds there are the following performance metrics: *Throughput* = 80 requests/second, *ResponseTime* = 2.1 seconds; (ii) the *OperationInstance* v2 is evaluated by solving the performance model in an analytic way, and the analytic solution reveals that there are the following performance metrics: *Throughput* = 40 requests/second, *ResponseTime* = 4.8 seconds. We recall that msec, sec, and min represent the enumeration values for the operation instance *metricUnit* element of the XML Schema.

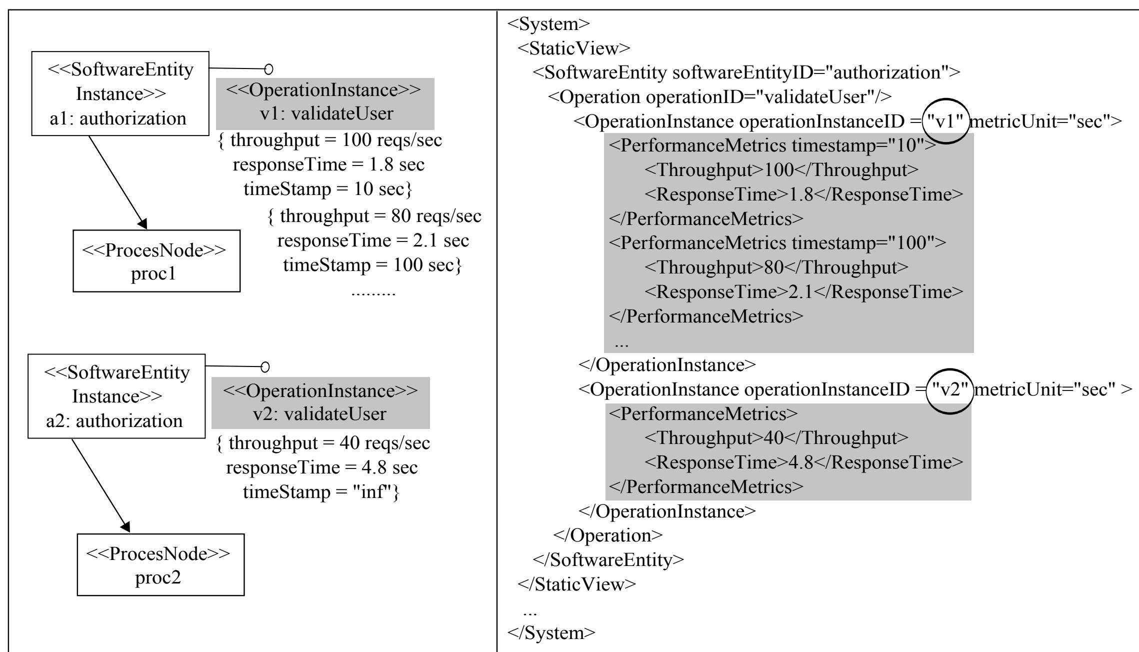


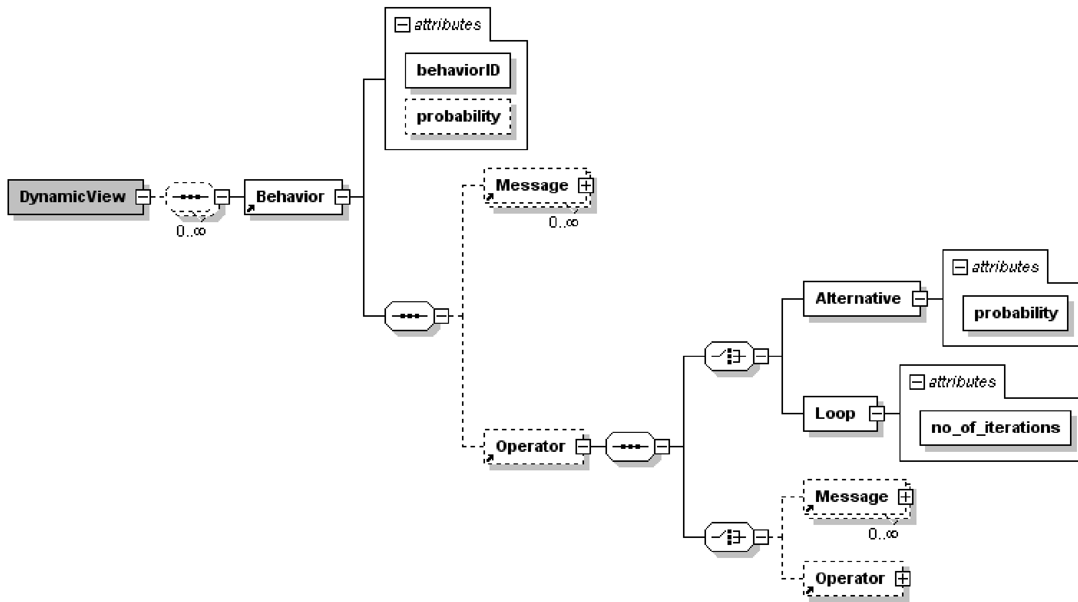
Figure A.5: An example of the *PerformanceMetrics* element.

A.2 DYNAMIC VIEW

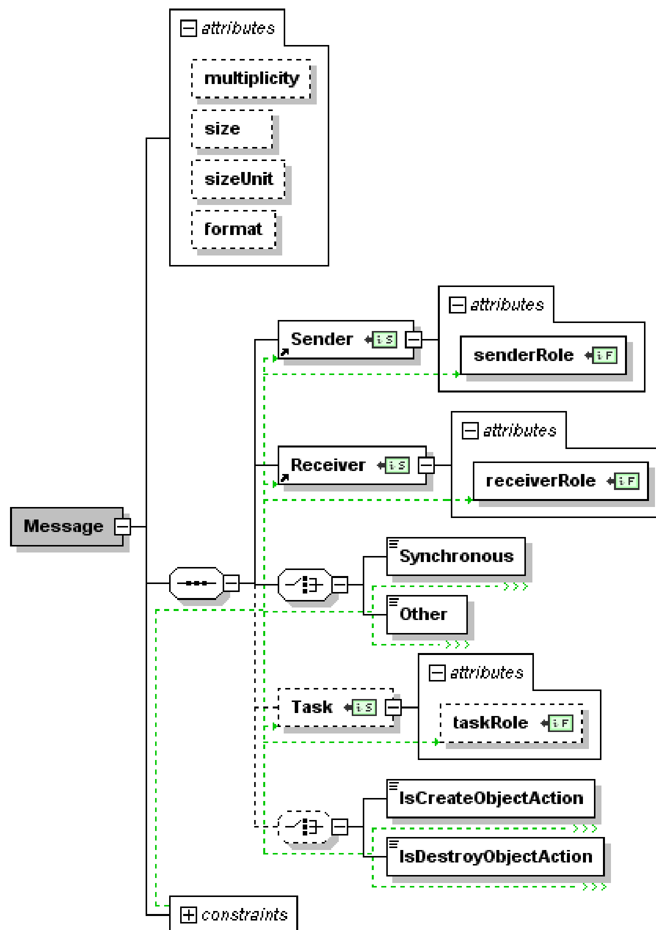
The *Dynamic View* (Figure A.6(a)) is made of *Behavior*(s). Each behavior has an identifier (*behaviorID*), and its execution *probability*. A *Behavior* contains either a set of *Message*(s) or an *Operator*.

As shown in Figure A.6(a), an *Operator* can be either a behavior *Alternative* with the *probability* it occurs, or a *Loop* with the number of iterations (*no_of_iterations*). An operator might contain *Message*(s) and optionally another nested *Operator*.

As shown in Figure A.6(b), a *Message* is described by the *multiplicity* of the communication pattern, the *size* and its *sizeUnit* (e.g. Kilobyte, Megabyte, Gigabyte), and the



(a) *Behavior* element.



(b) *Message* element.

Figure A.6: XML Schema - *Dynamic View*.

format (e.g. xml) used in the communication. Note that the *sizeUnit* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *Kb* (referring to Kilobyte), *Mb* (referring to Megabyte), *Gb* (referring to Gigabyte), and *other* (referring to any other size unit customized by the user). From a performance perspective, the size of messages is useful to detect antipatterns in message-based systems that require heavy communication overhead for sending messages with a very small amount of information, whereas the format of messages is useful when the sender of the message translates it into an intermediate format, and then the receiver parses and translates it in an internal format before processing it. The translation and parsing of formats could be time consuming, thus degrading the performance.

A message has a *Sender* identified by *senderRole*, a *Receiver* identified by *receiverRole*, and they both refer to a *softwareEntityInstanceID* playing such role. Additionally, it is possible to specify if the message is *Synchronous* or *Other*. We are not interested to asynchronous and reply messages, we flatten them in *Other*. Also in this case we can notice that the XML Schema we propose it is not meant to represent software systems, but to keep all the concepts providing useful information for the antipatterns. In fact, synchronous messages are of particular interest for the One-Lane Bridge antipattern, since it occurs when a set of processes make a synchronous call to another process that is not multi-threaded, hence only a few processes may continue to execute concurrently. Note that *Other* is intentionally added to enrich the vocabulary of the XML Schema, and it can be further detailed if new antipatterns are considered.

The *Task* determines the invocation of one operation, it is characterized by an identifier (*taskRole*) that refers to an *operationInstanceID* playing such role. Optionally there may be an attribute to specify the semantic of the message (i.e. *IsCreateObjectAction* or *IsDestroyObjectAction*) that allows the detection of frequent and unnecessary creations and destructions of objects belonging to the same software entity instance.

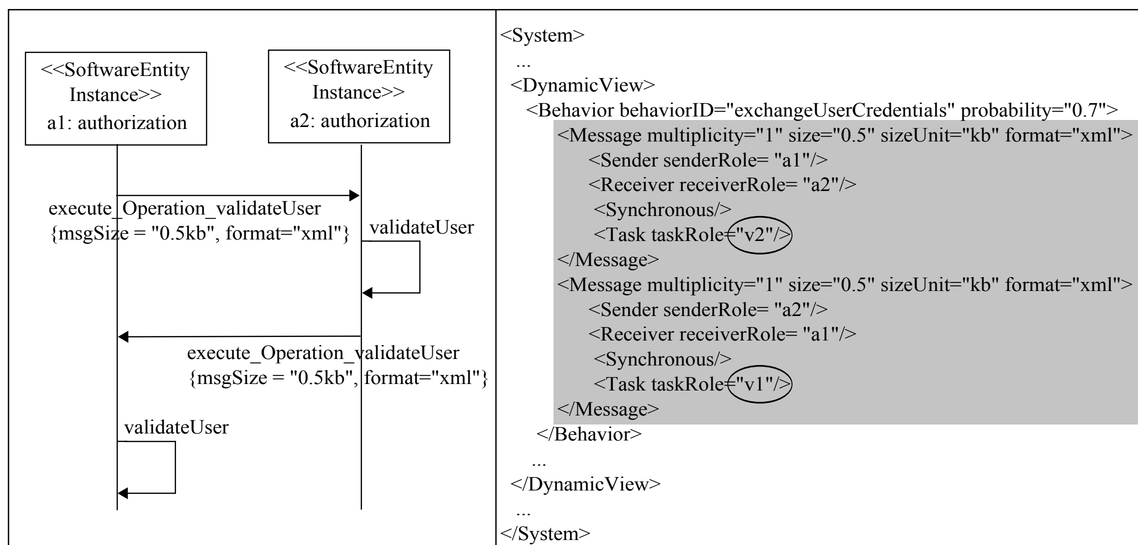


Figure A.7: An example of the *Behavior* element.

Figure A.7 shows an example of elements belonging to the Dynamic View of the XML Schema: the left side of the Figure gives a graphical representation of a software architectural model, whereas the right side reports an excerpt of the XML file (compliant to the XML Schema) representing the features of the software architectural model. In particular, Figure A.7 shows an example of the *Behavior* element. There are two software entity instances (a1 and a2) exchanging the user credentials, and the behavior is performed as follows. The software entity instance a1 sends a synchronous message (with a size of 0.5 kilobyte and xml format) to the software entity instance a2 by invoking the execution of the operation validateUser. Note that the *taskRole* attribute of Figure A.7 refers to the *operationInstanceID* of Figure A.5.

A.3 Deployment View

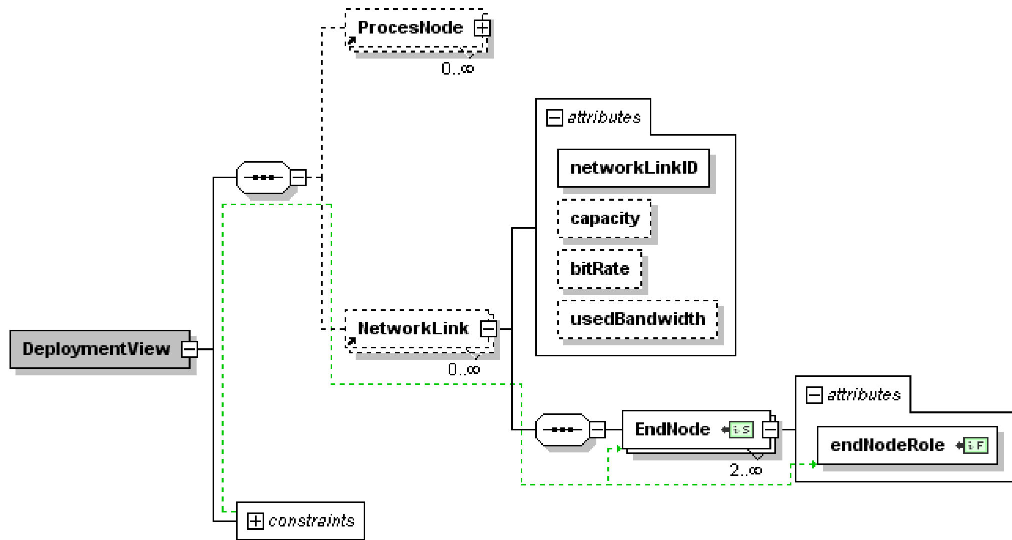
The *DeploymentView* (see Figure A.8(a)) is made of a set of processing nodes (*ProcesNode*), and optional *NetworkLink*(s) that enable the communication between the nodes.

As shown in Figure A.8(b), a processing node has an identifier (*procesNodeID*), and it contains a set of *DeployedInstance*(s). Each *DeployedInstance* has an identifier (*deployedInstanceRole*) that refers to a *softwareEntityInstanceID* already defined in the Static View.

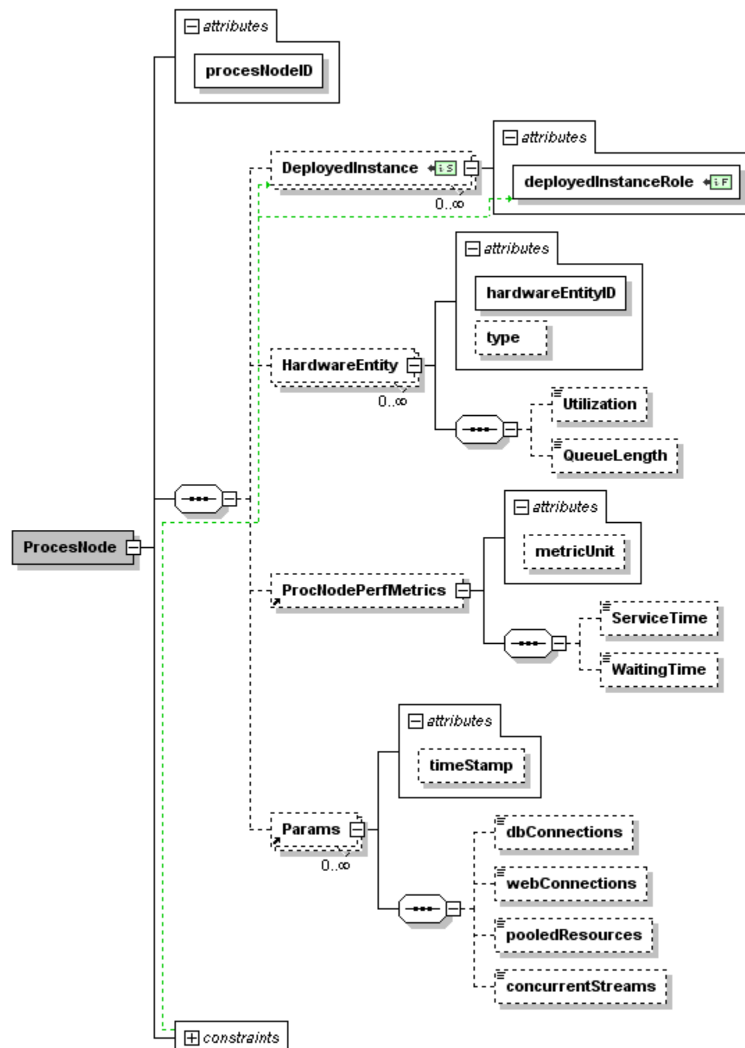
A processing node additionally contains a set of hardware entities (*HardwareEntity*). Each hardware entity has an identifier (*hardwareEntityID*), the *Utilization* and the *QueueLength* values, and the *type* indicating whether it is a *cpu* or a *disk*. In fact the *type* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *cpu* (referring to cpu devices), *disk* (referring to disk devices), and *other* (referring to any other type customized by the user). From a performance perspective, the distinction between cpu(s) and disk(s) is useful to point out the type of work assigned to a processing node by checking their utilization values (e.g. a database transaction uses more disk than cpu).

Specific performance metrics for a processing node are also defined, i.e. *ProcNodePerfMetrics*, such as its *ServiceTime* (e.g. the average processing time used to execute requests incoming to the processing node is 2 seconds), and its *WaitingTime* (e.g. the average waiting time for requests incoming to the processing node is 5 seconds). The *metricUnit* attribute denotes again the measurement unit adopted (e.g. micro seconds, seconds, minutes).

From the performance perspective, some additional information can be useful in this view. It may happens that while trying to run too many programs at the same time this introduces an extremely high paging rate, thus systems spend all their time serving page faults rather than processing requests. In order to represent such scenario we introduce a set of parameters, *Params*, that the processing nodes can manage: the number of database connections (*dbConnections*), the number of internet connections (*webConnections*), the amount of



(a) *NetworkLink* element.



(b) *ProcesNode* element.

Figure A.8: XML Schema - *Deployment View*.

pooled resources (*pooledResources*), and finally the number of concurrent streams (*concurrentStreams*). All these parameters are associated to a certain *timestamp* to monitor their trend along the time.

A *NetworkLink* (see Figure A.8(a)) has an identifier (*networkLinkID*), and two or more *EndNodes*. Each end node has an identifier (*endNodeRole*) that refers to *procesNodeID* playing such role. It optionally contains information about the network: the available bandwidth (*capacity*) denoting the maximum message size it supports, its utilization (*usedBandwidth*), and the *bitRate* used in the communication. Note that the *bitRate* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *Kbit/s* (referring to Kilobit/second), *Mbit/s* (referring to Megabit/second), *Gbit/s* (referring to Gigabit/second), and *other* (referring to any other bit rate unit customized by the user). From a performance perspective, the bit rate of the network is useful to detect antipatterns in message-based systems that require heavy communication overhead for sending messages with a very small amount of information.

Figures A.9, A.10 show some examples of elements belonging to the Deployment View of the XML Schema: the left sides of the Figures give graphical representations of a software architectural model, whereas the right sides report an excerpt of the XML files (compliant to the XML Schema) representing the features of the software architectural model.

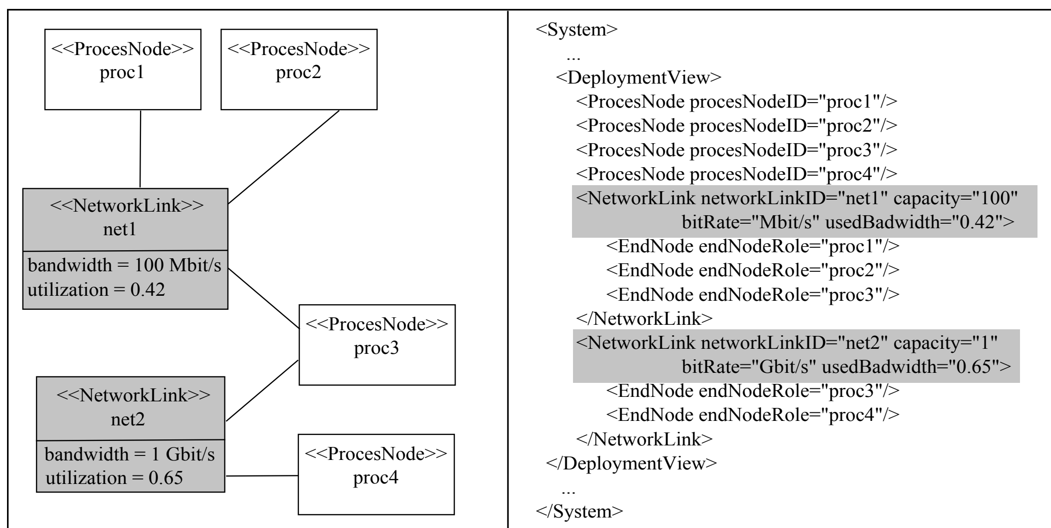


Figure A.9: An example of the *NetworkLink* element.

Figure A.9 shows an example of the *NetworkLink* element. There are four processing nodes, i.e. proc1, . . . , proc4, communicating through two network links: the *NetworkLink* net1 allows the communication among the processing nodes proc1, proc2, and proc3; the *NetworkLink* net2 allows the communication among the processing nodes proc3, and proc4. For each network link it is specified the bandwidth and the utilization; for example, net1 has a *capacity* of 100 with a *bitRate* expressed in Megabit/second and its *usedBandwidth* is equal to 0.42.

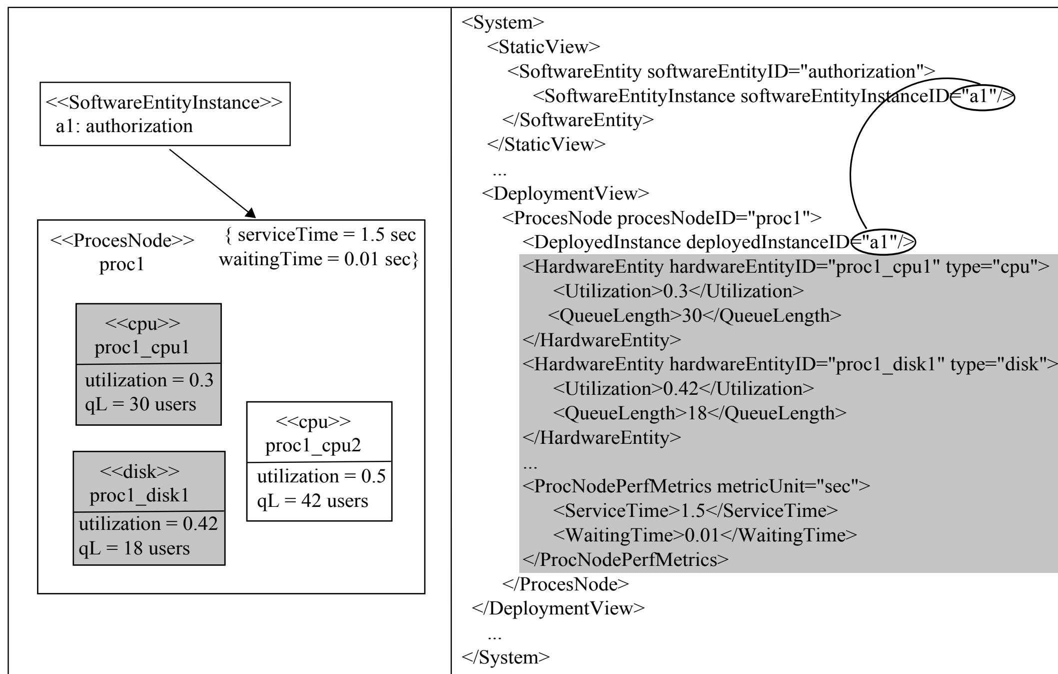


Figure A.10: An example of the *ProcesNode* element.

Figure A.10 shows an example of the *ProcesNode* element. There is a processing node (proc1) with three hardware entities, i.e. two cpus (proc1_cpu1, proc1_cpu2) and one disk (proc1_disk1). For each hardware entity it is possible to specify the queue length (qL) and the utilization; for example the proc1_cpu1 hardware entity reveals an average utilization of 0.3, and an average queue length of 30 users. Additionally, some performance metrics are evaluated for the processing node: the *serviceTime* keeps the time necessary to perform a task (i.e. 1.5 seconds in proc1), and the *waitingTime* stores how long incoming requests to the node must wait before being processed (i.e. 0.01 seconds in proc1).

A.4 CONCLUSION

In this Appendix we provided a generic data structure (an XML Schema) collecting all the architectural model elements that occur in the definitions of antipatterns [123] (such as software entity, hardware utilization, operation throughput, etc.). It represents a groundwork for the definition of antipatterns as logical predicates (see Section 3.4), thus to achieve a formalization of the knowledge commonly encountered by performance engineers in practice.

The XML Schema we defined obviously shares many concepts with existing software modeling languages (such as UML [12] and ADL [102]). However, it is not meant to be another software modeling language, rather it is strictly aimed at specifying the basic elements of performance antipatterns. Note that we only address performance antipatterns that can be defined independently of the notation adopted for the software modeling, in fact in our XML Schema we define elements that are independent from any particular modeling notation.

APPENDIX B

MODELING NOTATIONS

This Chapter briefly reviews the most used software (e.g. automata, process algebras, etc.) and performance modeling (e.g. queueing networks, generalized stochastic petri nets, etc.) notations. The introduction of modeling notations makes indeed the problem of interpreting the results of performance analysis quite complex as each modeling notation is intended to give a certain representation of the software system, expressed in its own syntax, and the performance results are necessarily tailored to that notation.

B.1 SOFTWARE ARCHITECTURAL MODEL

In the following we shortly review Automata [82], Process Algebras [104], Petri Nets [66], Message Sequence Charts [49], and Use Case Maps [17]. Unified Modeling Language (UML) [12] and Palladio Component Model (PCM) [22] are both software modeling notations used for the validation of the thesis approach, for more details please refer to Sections 5.1.1 and 5.2.1 respectively.

B.1.1 AUTOMATA

Automaton [82] is a simple mathematical and expressive formalism that allows to model cooperation and synchronization between subsystems, concurrent and not. It is a compositional formalism where a system is modeled as a set of states and its behavior is described by transitions between them, triggered by some input symbol.

More formally an automaton is composed of a (possibly infinite) set of states Q, a set of input symbols $\sum$ and a function $\delta : Q \times \sum \rightarrow Q$ that defines the transitions between states. In Q there is a special state $q_0 \in Q$, the initial state from which all computations start, and a set of final states $F \subset Q$ reached by the system at the end of correct finite computations [82]. It is always possible to associate a direct labeled graph to an automaton, called State Transition Graph (or State Transition Diagram), where nodes represent

the states and labeled edges represent transitions of the automata triggered by the input symbols associated to the edges.

There exist many types of automata: *deterministic* automata with a deterministic transition function, that is the transition between states is fully determined by the current state and the input symbol; *non deterministic* automata with a transition function that allows more state transitions for the same input symbol from a given state; *stochastic* automata which are non deterministic automata where the next state of the system is determined by a probabilistic value associated to each possibility. Besides, automata can also be composed through composition operators, notably the parallel one that composes two automata A and B by allowing the interleaving combination of A and B transitions.

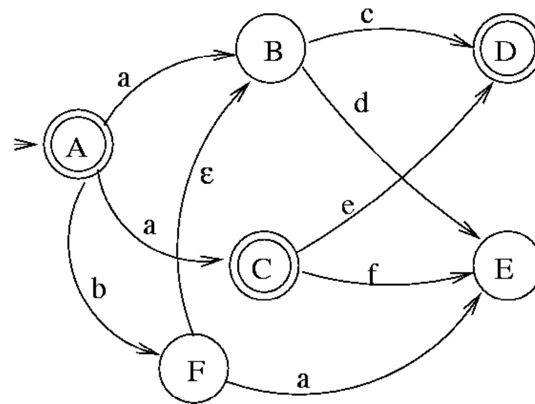


Figure B.1: A simple example of the *Automata* modeling notation.

Figure B.1 shows a simple example of the *Automata* modeling notation. It contains six states labeled A, B, C, D, E and F. The state labeled A is the initial one. The states A, C, and D are final. Starting in the initial states, the automaton processes a sequence of input symbols. In a given state, it checks if the next input symbol matches any of the labels of the transitions that go out of the state. A set of sequences of symbols gathered by pursuing all paths from initial states to the final ones is called the *language* accepted by the automaton.

B.1.2 Process Algebras

Process Algebras, such as Communicating Sequential Processes (CSP) [18] and Calculus of Communicating Systems (CCS) [104], are a widely known modeling technique for the functional analysis of concurrent systems. These are described as collections of entities, or processes, executing atomic actions, which are used to describe concurrent behaviors which synchronize in order to communicate.

Processes can be composed by means of a set of operators, which include different forms of parallel composition. Operators are provided by the language to construct processes out

of smaller processes [78]. A parallel composition operator is used to express concurrent execution and possible synchronization of processes. Another important operator realizes abstraction. Details of a specification which are internal details at a certain level of system description can be internalized by hiding them from the environment. Several notions of equivalence make it possible to reason about the behavior of a system, e.g. to decide whether two systems are equivalent.

More recent additions to the family of process algebras include the Π-calculus [105], the ambient calculus [36]. The semantics of these calculi is usually defined in terms of Labelled Transition Systems (LTS) [86] following the structural operating semantics approach. Moreover Process Algebra formalism is used to detect undesirable properties and to formally derive desirable properties of a system specification. Notably, process algebra can be used to verify that a system displays the desired external behavior, meaning that for each input the correct output is produced.

B.1.3 Petri Nets

Petri Nets (PN) are a formal modeling technique to specify synchronization behavior of concurrent systems. A PN [112] is defined by a set of places, a set of transitions, an input function relating places to transitions, an output function relating transition to places, and a marking function, associating to each place a non negative integer number where the sets of places and transitions are disjoint sets.

PN have a graphical representation: places are represented by circles, transitions by bars, input function by arcs directed from places to transitions, output function by arcs directed from transitions to places, and marking by bullets, called tokens, depicted inside the corresponding places. Tokens distributed among places define the state of the net. The dynamic behavior of a PN is described by the sequence of transition firings that change the marking of places (hence the system state). Firing rules define whether a transition is enabled or not.

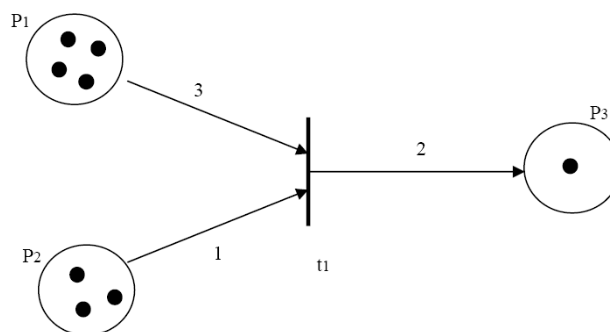


Figure B.2: A simple example of the *Petri Nets* modeling notation.

Figure B.2 shows a simple example of the *Petri Nets* modeling notation. It contains three places, i.e. P_1, P_2, and P_3, and one transition, i.e. t_1. Places contain four, three, and one tokens respectively. The transition is enabled, since in P_1 there are at least three (i.e. the weight of the arc connecting the P_1 place with the transition t_1) tokens, and in in P_2 there is at least one (i.e. the weight of the arc connecting the P_2 place with the transition t_1) token. The firing of the transition t_1 lead the system to evolve in a new configuration: the tokens of P_1 and P_2 places are respectively decreased of 3 and 1 units, whereas tokens in P_3 are increased of 2 units.

B.1.4 MESSAGE SEQUENCE CHARTS

Message Sequence Charts (MSC) is a language to describe the interaction among a number of independent message-passing instances (e.g. components, objects or processes) or between instances and the environment. This language is specified by the International Telecommunication Union (ITU) in [11]. MSC is a scenario language that describes the communication among instances, i.e. the messages sent, messages received, and the local events, together with the ordering between them. One MSC describes a partial behavior of a system. Additionally, it allows for expressing restrictions on transmitted data values and on the timing of events.

MSC is also a graphical language which specify two-dimensional diagrams, where each instance lifetime is represented as a vertical line, while a message is represented by a horizontal or slanted arrow from the sending process to the receiving one. MSC supports complete and incomplete specifications and it can be used at different levels of abstraction. It allows to develop structured design since simple scenarios described by basic MSC can be combined to form more complete specifications by means of high-level MSC.

B.1.5 USE CASE MAPS

Use Case Maps (UCM) [34] is a graphical notation allowing the unification of the system use (Use Cases) and the system behavior (Scenarios and State Charts) descriptions. UCM is a high-level design model to help humans express and reason about system large-grained behavior patterns. However, UCM does not aim at providing complete behavioral specifications of systems. At requirement level, UCM models components as black boxes, and at high-level design refines components specifications to exploit their internal parts.

Figure B.3 shows the basic elements of UCM notation. UCM represents scenarios through path scenarios: the starting point of a map corresponds to the beginning of a scenario. Moving through the path, UCM represents the scenario in progress till its end.

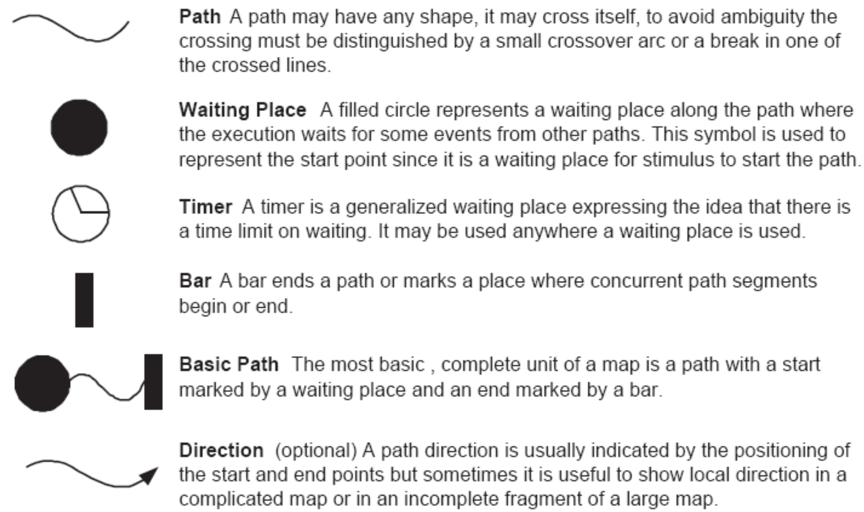


Figure B.3: Basic Symbols of the *Use Case Maps* modeling notation.

B.2 Performance Model

In the following we shortly review Markov Processes [87], Queueing Networks [88, 83, 92], Stochastic Process Algebras [77], Stochastic Timed Petri Nets [38, 97], and Simulation Models [20].

B.2.1 Markov Processes

A stochastic process is a family of random variables $X = \{X(t) : t \in T\}$ where $X(t) : T \times \Omega \to E$ defined on a probability space Ω, an index set T (usually referred as time) with state space E. Stochastic processes can be classified according to the state space, the time parameter, and the statistical dependencies among the variables $X(t)$. The state space can be discrete or continuous (processes with discrete state space are usually called chains), the time parameter can also be discrete or continuous, and dependencies among variables are described by the joint distribution function.

Informally, a stochastic process is a Markov process if the probability that the process goes from state $s(t_n)$ to a state $s(t_{n+1})$ conditioned to the previous process history equals the probability conditioned only to the last state $s(tn)$. This implies that a process is fully characterized by these one-step probabilities. Moreover, a Markov process is homogeneous when such transition probabilities are time independent. Due to the memoryless property, the time that the process spends in each state is exponential or geometrically distributed for the continuous-time or discrete-time Markov process, respectively.

Markov processes [87] can be analyzed and under certain constraints it is possible to derive the stationary and the transient state probability. The stationary solution of the

Markov process has a time computational complexity of the order of the state space E cardinality. Markov processes play a central role in the quantitative analysis of systems, since the analytical solution of the various classes of performance models relies on a stochastic process which is usually a Markov process.

B.2.2 Queueing Networks

Queueing Network (QN) models have been widely applied as system performance models to represent and analyze resource sharing systems [88, 83, 92]. A QN model is a collection of interacting service centers representing system resources and a set of customers representing the users sharing the resources. Its informal representation is a direct graph whose nodes are service centers and edges represent the behavior of customers' service requests.

The popularity of QN models for system performance evaluation is due to the relative high accuracy in performance results and the efficiency in model analysis and evaluation. In this setting the class of product form networks plays an important role, since they can be analyzed by efficient algorithms to evaluate average performance indices. Specifically, algorithms such as convolution and Mean Value Analysis have a computational complexity polynomial in the number of QN components. These algorithms, on which most approximated analytical methods are based, have been widely applied for performance modeling and analysis.

Informally, the creation of a QN model can be split into three steps: definition, that include the definition of service centers, their number, class of customers and topology; parameterization, to define model parameters, e.g., arrival processes, service rate and number of customers; evaluation, to obtain a quantitative description of the modeled system, by computing a set of figures of merit or performance indices such as resource utilization, system throughput and customer response time. These indices can be local to a resource or global to the whole system.

Extensions of classical QN models, namely Extended Queuing Network (EQN) models, have been introduced in order to represent several interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queues, memory constraints and simultaneous resource possession. EQN can be solved by approximate solution techniques [83, 92].

Another extension of QN models is the Layered Queuing Network (LQN) which allows the modeling of client-server communication patterns in concurrent and/or distributed software systems [136, 60]. The main difference between LQN and QN models is that in LQN a server may become client (customer) of other servers while serving its own clients requests. A LQN model is represented as an acyclic graph whose nodes are software entities (or tasks) and hardware devices, and whose arcs denote service requests (through synchronous, asynchronous or forwarding messages). A task has one or more entries

providing different services, and each entry can be decomposed in two or more sequential phases. A recent extension of LQN allows for an entry to be further decomposed into activities which are related in sequence, loop, parallel (AND fork/join) and alternative (OR fork/join) configurations forming altogether an activity graph. LQN models can be solved by analytic approximation methods based on standard methods for EQN with simultaneous resource possession and Mean Value Analysis or they can be simulated.

B.2.3 Stochastic Process Algebras

Several Stochastic extensions of Process Algebras (SPA) have been proposed in order to describe and analyze both functional and performance properties of software specifications. Among these we consider PEPA (Performance Evaluation Process Algebra) [65], and EMPA (Extended Markovian Process Algebra) [27, 28, 25]. PEPA is of worth interest, since it extends classical process algebras such as CSP and CCS by introducing probabilistic branching and timing of transitions. $EMPA_{gr}$ [24] was introduced in order to easy the use of SPA as software architectural model notation, and in particular it represents the algebra on which it is based an Architectural Description Language (ADL) called Æmilia [26].

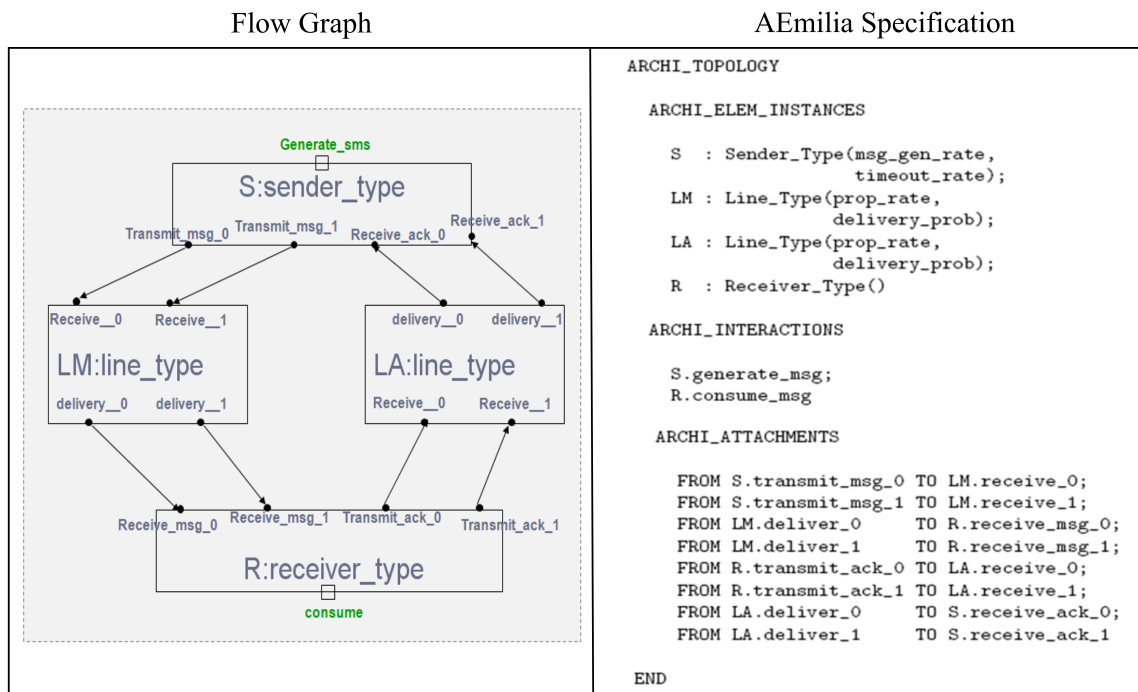


Figure B.4: A simple example of the *Æmilia* modeling notation.

Æmilia aims at facilitating the designer in the process algebra-based specification of software architectures, by means of syntactic constructs for the description of architectural components and connections. Æmilia is also equipped with checks for the detection of

possible architectural mismatches. Moreover for Æmilia specifications a translation into the Queueing Network (QN) performance models (see Section B.2) has been proposed in order to take advantage of the orthogonal strengths of the two formalisms: formal techniques for the verification of functional properties for Æmilia as SPA in general, and efficient performance analysis for Queueing Networks.

Figure B.4 shows a simple example of the *Æmilia* modeling notation. On the left side of the Figure the flow graph representation of a software system (i.e. the Alternating Bit Protocol) is shown; on the right side of the Figure an excerpt of the Æmilia specification is reported. In particular, only the topology of the system (i.e. architectural element instances, interactions, and attachments) is shown, whereas the behavior of the different element types is not reported for sake of readability [1].

B.2.4 Stochastic Timed Petri Nets

Stochastic Timed Petri Net (STPN) are extensions of Petri nets. Petri nets can be used to formally verify the correctness of synchronization between various activities of concurrent systems. The underlying assumption in PN is that each activity takes zero time (i.e. once a transition is enabled, it fires instantaneously). In order to answer performance-related questions beside the pure behavioral ones, Petri nets have been extended by associating a finite time duration with transitions and/or places (the usual assumption is that only transitions are timed) [38, 97, 83].

The firing time of a transition is the time taken by the activity represented by the transition: in the stochastic timed extension, firing times are expressed by random variables. Although such variables may have an arbitrary distribution, in practice the use of non memoryless distributions makes the analysis unfeasible whenever repetitive behavior is to be modeled, unless other restrictions are imposed (e.g. only one transition is enabled at a time) to simplify the analysis.

The quantitative analysis of a STPN is based on the identification and solution of its associated Markov Chain built on the basis of the net reachability graph. In order to avoid the state space explosion of the Markov Chain, various authors have explored the possibility of deriving a product-form solution for special classes of STPN. Non polynomial algorithms exist for product-form STPN, under further structural constraints. Beside the product-form results, many approximation techniques have been defined [38].

A further extension of Petri Nets is the class of the so called Generalized Stochastic Petri Nets (GSPN), which are continuous time stochastic Petri Nets that allow both exponentially timed and untimed (or immediate) transitions [96]. Immediate transition fires immediately after enabling and have strict priority over timed transitions. Immediate transitions are associated with a (normalized) weight, so that, in case of concurrently enabled imme-

[1] The example is widely illustrated in [50]

diate transitions the choice of the firing one is solved by a probabilistic choice. GSPN admit specific solution techniques [38].

B.2.5 Simulation Models

Besides being a solution technique for performance models, simulation can be a performance evaluation technique itself [20]. It is actually the most flexible and general analysis technique, since any specified behavior can be simulated. The main drawback of simulation is its development and execution cost. Simulation of a complex system includes the following phases:

- building a simulation model (i.e., a conceptual representation of the system) using a process oriented or an event oriented approach;
- deriving a simulation program which implements the simulation model;
- verifying the correctness of the program with respect to the model;
- validating the conceptual simulation model with respect to the system (i.e. checking whether the model can be substituted to the real system for the purposes of experimentation);
- planning the simulation experiments, e.g. length of the simulation run, number of run, initialization;
- running the simulation program and analyzing the results via appropriate output analysis methods based on statistical techniques.