TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2013

27

Diego Carmelo Pérez Palacín

Extra Functional Properties Evaluation of Self-managed Software Systems with Formal Methods

Departamento

Informática e Ingeniería de Sistemas

Director/es

Merseguer Hernáiz, José Javier

http://zaguan.unizar.es/collection/Tesis

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

Universidad Zaragoza

Tesis Doctoral

EXTRA FUNCTIONAL PROPERTIES EVALUATION OF SELF-MANAGED SOFTWARE SYSTEMS WITH FORMAL METHODS

Autor

Diego Carmelo Pérez Palacín

Director/es

Merseguer Hernáiz, José Javier

UNIVERSIDAD DE ZARAGOZA

Informática e Ingeniería de Sistemas

2013

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

Extra Functional Properties Evaluation of Self-managed Software Systems with Formal Methods

Diego Pérez Palacín

TESIS DOCTORAL

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Supervisor: José Javier Merseguer Hernáiz

December 2012

Preface

Today software applications are often intended to operate in dynamic contexts. These dynamic contexts can be expressed in terms of changes in the application execution environment, changes in the application requirements, changes in the workload, or changes in anything the software can perceive. These software applications are not restricted to a single domain but dynamic contexts can be found in many software domains such as: embedded systems, service-oriented architectures, clusters for high performance computing, mobile devices and networking software.

The existence of dynamic contexts discourages engineers from the development of software that is not able to change, in some way, its execution to leverage the current context at any moment. Then, with the intention of satisfying requirements, the software must include mechanisms to change its configuration. Nevertheless, since context changes are frequent and go through many devices, human intervention to manually change the software configuration is not a feasible solution. To face this challenge, the Sofware Engineering community came up with new paradigms to enable the development of software that deals with dynamic contexts in an automatic manner; for example Autonomic Computing and self-* software approaches. In these approaches the software manages the mechanisms to change the configuration, without requiring human intervention. Among the most general manners to refer to self-* software are the terms self-adaptive and self-managed software.

An essential aspect of self-adaptive software is to plan its adaptations. Adaptation plans determine how the system will adapt and the suitable moments to trigger these adaptations. There is a vast set of situations for which self-adaptivity is a solution. One of these situations is to keep the system satisfying its extra functional requirements, such as Quality of Service (QoS) and energy consumption. This thesis investigates these situations.

One of the contributions of this thesis is to settle QoS- and energy-aware self-adaptive systems into an architecture. For this purpose, a 3-layer architecture of reference for self-managed systems guides our research. The goodness of using a reference architecture is that it easily shows up new challenges for the design of this kind of systems. Of course, adaptation planning is one of the activities that the architecture considers.

Another contribution of this thesis is to propose methods to create adaptation plans. Formal methods play an essential role in this activity since they help to study extra functional properties of the system under different configurations. Markovian Petri nets are used for this analysis. Once the adaptation plan has been created, we also investigate the evaluation of the QoS and energy consumption of self-adaptive systems with formal methods. Thence, we

contribute to the QoS analysis community -whose research cares about analysis of software systems quality properties- by investigating a particularly new and complex type of software systems. To carry out this analysis it is required to model dynamic changes in the context, for which a variety of formal methods can be used: Markov modulated Poisson processes to estimate the dynamic workload parameters, and hidden Markov models to predict the environment state. These models were used together with Petri nets to evaluate and obtain results about QoS and energy consumption of self-adaptive systems.

During the research work, we advertised that adaptability is a system property not as easily quantifiable as energy consumption or QoS properties -such as the response time. Research was done in this direction and, as a result, this thesis contributes to quantify the adaptability by proposing a set of adaptability metrics for service-based systems. Moreover, we studied whether there is a correlation between the adaptability of a system and other of its extra functional properties values.

For achieving these contributions, this thesis makes an intensive use of models and model transformations; task for which best practices of the Model Driven Engineering (MDE) research field are followed. The research work of this thesis in the MDE field resulted in: enhancement of the modeling power of an already proposed software modeling language, and transformation methods from two software modeling languages to stochastic Petri nets.

Contents

1 Introduction **1**
1.1 Outline 4

I Evaluation of Self-managed Systems **7**

2 Reference Architecture for Self-managed Systems **9**
2.1 Enabling software to face the unknown 11
2.2 Self-adaptive systems manage ignorance 12

3 Self-adaptation for Performance Engineering: Goal Management **15**
3.1 Problem description 15
3.2 3-Layer architecture for open-world software 16
3.2.1 Component control 16
3.2.2 Change management 17
3.2.3 Goal management 19
3.3 Generation of strategies 19
3.4 Example 21
3.4.1 Strategy generation 22
3.5 Conclusion 26
3.6 Related work 27

4 Self-adaptation for Performance Engineering: Change Management **31**
4.1 Problem description 31
4.2 A formal model for SOA providers 32
4.2.1 Hidden Markov models 32
4.2.2 HMM representation of SOA providers 34
4.3 State prediction and configuration adaptation 35
4.3.1 Prediction of the provider state 35
4.3.2 Adaptations based on state predictions 36
4.4 Integrating the adaptive configurations into an architectural solution 40
4.5 Conclusion 41
4.6 Related work 42

5 Self-adaptation for Energy Conservation **43**
5.1 Motivation . . . 43
5.2 Architecture . . . 46
5.3 Workload modeling . . . 48
5.4 Energy modeling and analysis . . . 51
5.5 Performance and energy trade-off . . . 54
5.5.1 Generation of basic-plan . . . 54
5.5.2 Reconfiguration rate mitigation . . . 55
5.5.3 Petri net model of a plan . . . 58
5.5.4 The Petri net for trade-off evaluation . . . 59
5.6 Deployment and evaluation . . . 60
5.6.1 Evaluation framework . . . 60
5.6.2 Example of evaluation: relay mail server . . . 62
5.7 Experimenting with variable workload . . . 65
5.8 Conclusion . . . 67
5.9 Related work . . . 68

6 Workload Modeling for Self-adaptive Software **71**
6.1 Motivation . . . 71
6.2 MAP's and MMPP's . . . 72
6.2.1 MMPP fitting from a workload trace . . . 74
6.2.2 GSPN workload model . . . 77
6.3 Modeling transient time between states . . . 78
6.3.1 Problem statement . . . 78
6.3.2 Setting parameters of workload model . . . 78
6.3.3 GSPN model for the transient time . . . 81
6.4 Comprehensive workload model . . . 83
6.5 Experimental analysis . . . 85
6.5.1 Results discussion . . . 87
6.6 Conclusion . . . 88
6.7 Related work . . . 89

7 Measuring and Correlating System Adaptability **91**
7.1 Problem statement . . . 91
7.2 Architectural adaptability quantification . . . 92
7.2.1 Architectural assumptions . . . 92
7.2.2 Adaptability metrics . . . 94
7.3 Relating adaptability to a system quality . . . 96
7.4 Example . . . 100
7.4.1 Computation of system qualities . . . 102
7.4.2 Relation of adaptability to availability and cost . . . 104
7.5 Relating quality requirements . . . 106
7.5.1 Graphical representation . . . 107
7.6 Conclusion . . . 107
7.7 Related work . . . 109

II Model-Driven Engineering for QoS Evaluation **111**

8 Model To Model Transformations: From CSM To GSPN **117**
8.1 Problem description . . . 117
8.2 Core Scenario Model . . . 118
8.3 CSM meta-classes translation . . . 119
8.3.1 Step translation . . . 120
8.3.2 Resource translation . . . 121
8.3.3 PathConnections translation . . . 122
8.3.4 Workload translation . . . 125
8.4 Tool development . . . 126
8.4.1 Tool design . . . 126
8.4.2 The algorithm for the translation . . . 130
8.4.3 Remarks on the analysis of the resulting GSPN . . . 133
8.5 Example of system analysis . . . 135
8.5.1 Qualitative properties analysis . . . 136
8.5.2 Quantitative properties analysis . . . 137
8.5.3 LQN and GSPN results comparison . . . 142
8.6 Conclusion . . . 143

9 Model To-Model Transformations: D-KLAPER and GSPNs **145**
9.1 Motivation . . . 145
9.2 The D-KLAPER intermediate model . . . 146
9.2.1 Metamodel extension . . . 148
9.3 The model-driven framework for reactive systems . . . 150
9.3.1 The basic methodology . . . 150
9.3.2 UML state machines as reactive systems models . . . 152
9.3.3 Transforming UML state machines into D-KLAPER models . . . 153
9.3.4 Transforming D-KLAPER models into Petri nets . . . 154
9.4 Example application . . . 155
9.4.1 Structural specification . . . 155
9.4.2 Reactive specification . . . 157
9.4.3 Translation into D-KLAPER models . . . 158
9.4.4 Translation into Petri nets and evaluation . . . 160
9.5 Conclusion . . . 162

10 Conclusion **167**

Relevant Publications Related to the Thesis **168**

A Generalized Stochastic Petri Nets **173**

B GSPN Composition **175**
B.1 MLGSPN Composition . . . 175

Bibliography **179**

Acronyms **190**

Chapter 1

Introduction

Today software applications are often intended to operate in dynamic contexts. These dynamic contexts can be expressed in terms of changes in the application execution environment, changes in the application requirements, changes in the workload, or changes in anything the software can perceive. This feature of software applications is not restricted to a single domain but dynamic contexts can be found in many software domains such as: embedded systems, service-oriented architectures, clusters for high performance computing, mobile devices and networking software. As an example, consider a service-based software application or architecture (SOA) made of multiple required third-party services. These third-party services are out of hand of the application because they are not managed by the same institution. Therefore, they can evolve in unforeseen manners while the application properties, such as its quality of service, strongly depend on their behavior.

The existence of dynamic contexts discourages engineers from developing software that is not able to change, in some way, its execution to accommodate the current context at any moment. In the previous example, a third-party service can change its properties and then it can become no longer suitable for satisfying application requirements. A solution for this situation passes through changing the application by replacing the degraded or diminished service by another one that allows meeting its requirements.

Then, with the intention of satisfying requirements, the software must provide mechanisms to change its configuration. Nevertheless, since context changes are frequent and go through many devices, human intervention to manually change the software configuration is not a feasible solution. To face this challenge in modern software, the Software Engineering community came up with new paradigms that enable the development of software that deals with dynamic contexts in an automatic manner; for example Autonomic Computing [KC03, HM08] and self-* software approaches. In these approaches is the same software who manages the mechanisms to change the configuration, without the need of human intervention as a result.

The term self-* embraces many properties, some of them are: self-configuring, self-optimizing, self-healing or self-protecting [KC03]. A software application that implements one of these properties is expected to manage a subset of concerns. To generally refer to

software systems that can in some way control and modify their own behavior, more general level of autonomic systems terms are: self-governing, self-managed, self-organizing or self-adaptive [ST09, KC03, OGT+99]. In this thesis, to refer to the most general view of autonomic systems, self-adaptive and self-managed terms are used, having the same meaning and therefore being interchangeable. One of the first definitions given for self-adaptive software was provided by DARPA in [DAR, Lad99, Lad00], and dates from 1997: *Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible*.

Due to all the intrinsic actions of self-adaptive systems -such as monitor its execution context, evaluate its behavior in that context and change it when necessary- self-adaptive software is not easy to design and build from scratch. Consequently, to help to construct self-adaptive systems, during the past years there have been proposed several frameworks. Examples are: MAPE-K structure of an autonomic element [KC03] (Monitor, Analyze, Plan, Execute, Knowledge), CADA autonomic control loop [DDF+06, CLG+09] (Collect, Analyze, Decide, Act), the adaptation methodology in [OGT+99] and a three-layered architectural approach for self-managed systems [KM07]. Part of the research done in this thesis is built up using these frameworks. The most studied and explored framework has been the three layer architecture for self-managed systems. It has eased the work of self-adaptive systems study since it easily shows up the new challenges in their design.

The above mentioned frameworks share common aspects. An essential one is to *plan* adaptations. Adaptation plans determines how the system will adapt and the suitable moments to trigger these adaptations. For example, the planning process is included in one of the layers of the architecture in [KM07]; as an explicit activity in the autonomic control loop in [KC03]; and also the methodology in [OGT+99] recognizes planning as a vital aspect of self-adaptive systems and differentiates between two types of planning: adaptation planning and observation planning (the need to observe the context is another common aspect of self-adaptive systems).

Thinking about “dynamic context” in broad sense, we can find a huge amount of software systems that come under this perspective. Besides, the growth of ubiquitous and pervasive computing just makes this amount increase [Sat01]. So, there is a vast set of situations for which self-adaptation is a solution; some examples are: survival-critical systems that adapt by self-healing when a problem threatens them to stop operating, web systems that adapt by self-protecting when they detect an attack, or mobile devices that adapt to optimize their network connectivity and bandwidth. Among all these situations, this thesis concentrates on using self-adaptation to keep the system satisfying its extra functional requirements [CNYM99] -such as Quality of Service (QoS) and energy consumption- under different environment conditions.

The software QoS analysis research community has cared about software QoS assessment for the last decades. Within this community, formal methods are a pillar for coping with day-to-day QoS evaluation challenges, and large research advances in the formal method theory have been achieved while pursuing this goal. Nevertheless, the nature of self-adaptive systems and their dynamic contexts cause that some parts of QoS evaluation approaches can hardly be completely reused; or, in case of being directly applied, the evaluation results can

be very far from the actual QoS. Thence, the QoS evaluation of self-adaptive systems requires new methods and techniques that take into account the changes in the context; and even more, it is likely that these methods have to be applied not only during development but also during software execution. However, due to the novelty of the self-adaptive software systems and the Software Engineering methodologies to construct them, these methods and techniques for QoS evaluation are still a research challenge. To obtain new contributions, the work in this thesis leans on the research advances done by the software QoS analysis community and on their formal methods usage. This work contributes by studying new formal models, not considered so far, that can be able to represent dynamic contexts and configuration changes in software systems.

For the QoS analysis, this thesis contributes to two fields. The first field is the Software Performance Engineering (SPE) [SW02b] and we contribute by investigating the adaptation plans generation, the adaptation decisions, and the evaluation of performance-aware self-adaptive software systems. The second field is the evaluation of *availability* property in software systems. This property defines *the probability of a system being available to service requests*. We investigated the evaluation of systems that must maintain an agreed level of *availability*. The studied adaptations in these systems are those ones triggered by changes in the received workload that overload the system and make it unavailable for users.

An extra functional property that is becoming more and more important for software service provider companies is the energy consumption of computing resources. As shown in [DKL+08, Ran10], the interest towards efficient use of technology is motivated by some alarming trends. For example, computing equipment in the U.S. alone is estimated to consume more than 20 million giga-joules of energy per year, IT analysis IDC [IDC] estimates the total worldwide spending on power management for enterprises was likely 40 billion dollars in 2009, and large computing infrastructures consumed in the U.S. the 1.5% of all electrical power and it grows at an annual rate of 12% [CJH+11]. However, there is room for power reduction: since not all computing resources are required all the time, but hardware infrastructures are over-provisioned to cope with worst-case scenarios, infrastructure self-adaptation is a way to save energy. In this thesis, we research in that direction by defining adaptation plans to manage computing infrastructures with the aim of spending as less energy as possible while satisfying the rest of its requirements.

From previous paragraphs, there can be noticed that system workload is an important aspect; it plays a role in the evaluation of system performance, achievable availability or power demand. The reason is that the optimal configuration of a system is different depending on the workload it receives. Then, to perform a model-based system evaluation it is necessary to create and attach a workload sub-model. For an accurate analysis of software systems, this sub-model cannot represent that the workload is constant because it is far from being stable in reality. In addition, when analyzing Internet services or networked systems, this workload presents high variability and *burstiness*, i.e., irregular spikes of congestion. This aspect must be present when analyzing self-adaptive systems. Even more, since the system starts the adaptation when a burst of requests is near to arrive, it is not only necessary that the sub-model accounts for bursty and regular periods but also the transient intervals between them. In this thesis, we refine workload modeling and we propose a model that takes into account the transient periods. Therefore, we can increase the accuracy of results obtained

from model-based analysis.

Formal methods have been shown during the last years as a reliable analysis paradigm for all of the above mentioned software properties. However, due to the novelty of self-adaptive software, it is a hot topic and the research community is stepping forward in the use of formal methods to evaluate these properties at present. In this thesis, we lean on the great background in software analysis with formal methods and we use them to both construct the adaptation plans and decide the appropriate moments for adaptations. Besides, we also propose formal models to evaluate the self-adaptive system behavior. Obviously, to carry out that evaluation it is required to model dynamic changes in the context, for which a variety of formal methods are used; such as Markov modulated Poisson processes to estimate the dynamic workload parameters, and hidden Markov models to predict the environment state. These models are used together with stochastic Petri nets to evaluate and obtain results about QoS and energy consumption of self-adaptive systems.

During the research work, we found out that QoS properties such as response time or energy consumption were quantifiable. However, adaptability property was not. Thence, research has been done in that direction. As a result, this thesis proposes a set of metrics to quantify the adaptability of service-based self-adaptive systems. Moreover, it is studied whether there is a correlation between the quantified adaptability of a system and its extra functional properties values. The theories that came out of this investigation have been implemented into a tool that automatically executes the study receiving as input a software architectural design.

As the reader can deduce from this introduction, thesis contributions make an intensive use of models, model-to-model transformations and model-based analysis. For model-to-model transformations, we use the state-of-the-art proposals of the Model Driven Engineering (MDE). Recent studies have realized that, due to amount of software design languages and analyzable languages, the number of model-to-model transformations required to translate any software design model to any analyzable model increased excessively. To solve it, the use of intermediate models and transformations in two steps have been proposed by the research community: first from the design model to the intermediate model, and later from the intermediate model to the analyzable model. In this thesis, we contribute to these challenges by: proposing transformations from two intermediate languages to stochastic Petri nets, and extending the modeling power of one of these intermediate languages to allow it to model characteristics of reactive systems, category to which most self-adaptive systems belong.

1.1 Outline

The rest of this thesis comprises eight chapters, which are grouped into two parts, and a conclusion. The balance is the following:

- The first part comprises six chapters. The first of these chapters -Chapter 2- presents the 3-layer architecture of reference for self-managed systems we use in the course of the thesis. Chapters 3 and 4 instantiate the reference architecture for performance-aware self-adaptive service-oriented systems. In the former, it is investigated the uppermost layer of the architecture and adaptation strategies are generated using Generalized

Stochastic Petri Nets (GSPN) [AMBC+95]. In turn, the latter investigates the middle layer, which is in charge of deciding whether to adapt the system, and it uses the theories of Hidden Markov Models (HMM). In Chapter 5 the 3-layer reference architecture is instantiated for energy savings. Again, GSPNs are utilized to create the adaptation plans. In Chapter 6, we identify a weakness in the modeling of bursty workloads and we investigate an accurate model for its representation based on GSPNs. Finally, in Chapter 7 we research the quantification of adaptability property; being the outcome of that research twofold: a set of metrics to measure the adaptability of software systems and a study about the correlation between adaptability values and QoS values.

- The second part comprises an introduction and two chapters that accomplish investigation in model to model transformations. The introduction presents the motivation and benefits of MDE paradigm. It shows up a weak point of applying MDE from software design to formal models and also explains the proposal of the research community to solve the weakness. After, in Chapter 8 we follow the community proposal by creating a translation theory between the Core Scenario Model (CSM) and GSPN and a tool that implements that theory. In Chapter 9, we extend the modeling power of a Dynamic Kernel Language for PErformance and Reliability analysis (D-KLAPER) to represent reactive software (most self-adaptive systems are also reactive systems). In that chapter, we propose the theory to translate UML state machine diagrams into the extended D-KLAPER, and to translate the extended D-KLAPER into GSPN. Finally, it is shown an example regarding the use of D-KLAPER or performance performance evaluation of service-based software systems.

- Chapter 10 presents the conclusions of this thesis. It summarizes the contributions and identifies future work and research directions.

Part I

Evaluation of Self-managed Systems

Chapter 2

Reference Architecture for Self-managed Systems

Understanding all the processes under self-adaptive systems execution is complex. Also its development has some intrinsic characteristics that make it a process harder than the development of non-adaptive systems.

To relieve developers of the most general thinking about the software construction, there have been proposed some frameworks for self-adaptive software design. These frameworks include the general tasks that self-adaptive software has to execute. Four of the most important frameworks proposed are: MAPE-K structure of an autonomic element [KC03] (Monitor, Analyze, Plan, Execute, Knowledge), CADA autonomic control loop [DDF$^+$06] (Collect, Analyze, Decide, Act), the adaptation methodology in [OGT$^+$99] and a three-layered architectural approach for self-managed systems [KM07, KM09].

The work in this thesis uses the three-layered architectural approach in [KM07, KM09]. This architecture was not built from scratch but based on solid pillars. It was inspired by architectural approaches already in the robotic research area -field which holds a large experience in the construction of autonomous systems-, and concretely on the Gat's architectural description in [GBMP97].

From that starting point, we tuned the concepts in the three layer architecture for self-managed systems. Our purpose is the construction of software whose adaptations are triggered to keep satisfied extra functional requirements -such as QoS and energy consumption-under different execution contexts. The general view of the architecture is shown in Figure 2.1. A layer by layer description is in next paragraphs, while in next chapters we propose fine-grained instances of this general view for each one of the challenges dealt with in this thesis.

Component control This layer is made of the components that accomplish the software application functionality. It also includes *sensors and monitors* that track the *status* of both the system and environment. Some examples of monitored elements can be: component performance, service providers performance, service providers availability, system load, hardware

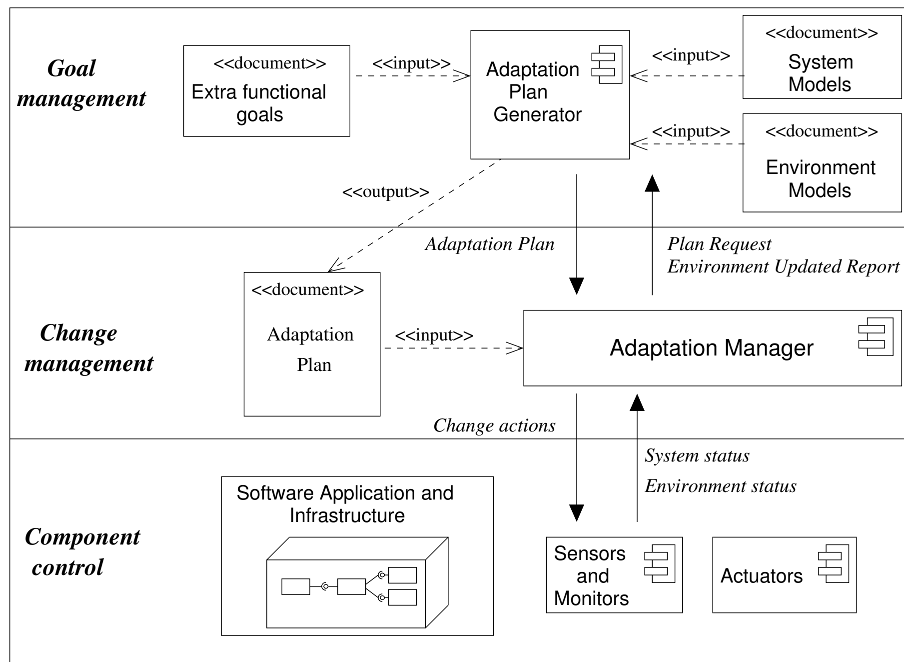


Figure 2.1: General architecture following the 3-layer framework for self-managed systems

status (e.g,. switched on, switched off, broken device). This sensed status information is sent to the upper layer. Besides, this layer includes *actuators*, which are the elements that effectively adapt the system execution. These actuators operate when they receive *change actions* from its upper layer. Some examples of elements onto which an actuator can operate are: software components with multiple execution configurations, replaceable software components, and hardware infrastructure. The *software application and infrastructure* comprises all the elements that allow the software to run its functionality, in which are included the monitored elements (i.e., those for which there is a sensor connected) as well as the managed elements (i.e., those for which there is an actuator connected).

Change management The objective of this layer is to keep the system executing in the most appropriate configuration to satisfy application's goals. To achieve this objective, the main task of this layer is to decide *change actions* to adapt the system based on the *status* reports received from the lowest layer. The software entity that executes the decision task is called *adaptation manager*. To perform this decision quickly, it has stored a pre-computed *adaptation plan* that contains information to guide the adaptations. In case that, in any point in time, the stored adaptation plan does not suffice to cope with the reported status, this layer can *request for a new plan* to the uppermost layer. To obtain a new adaptation plan that

considers the current execution environment, this layer informs the uppermost layer about what it was found in the environment. This notification is done through an *environment updated report*. Moreover, this layer can also receive a new adaptation plan that has been proactively created by the uppermost layer; i.e., the adaptation manager had not made any request for a new plan.

Goal management The objective of this layer is to produce adaptation plans that allow the system to satisfy its extra-functional goals under changing environments. The software entity that produces these plans is called *adaptation plan generator* and its execution requires information regarding: *extra-functional goals* to achieve, *application models* (e.g., software behavioral models and hardware platform models) and *environment models*. All these models are subject to change, and a change in a model can make the created adaptation plan no longer suitable. Therefore, after a change in the models, it should be triggered the generation of a new adaptation plan and its subsequent delivery to the change management layer. For example, changes in environment models are done when an *environment updated report* is received from the lower layer.

2.1 Enabling software to face the unknown

The type of self-adaptive systems investigated in this thesis have to face unknown situations and unpredictable environments. When software execution environment changes, the first challenge for the system is to realize that something in the environment has actually changed. This environmental change awareness is not always an easy task, and it becomes more difficult for slight environmental changes or when the change happens slowly. If the environment change is realized, next step for the software is to acquire knowledge about what the new execution environment is; i.e., not only *sensors reports indicate that something has changed in the environment* but *sensors reports indicate that the execution environment has changed from context "a" to context "b"*. After, software decides either to continue in the same configuration (in case that the current configuration still works well in the new environment) or change to another one (labour of the intermediate layer of the presented architecture). If it is decided to change, software has to find out the new configuration. For cases in which the occurrence of the new environment is *predictable* beforehand, the software can have stored a set of pre-computed reconfiguration actions to take when the environmental modification takes place. These actions are stored in an *adaptation plan*, which can guide the adaptations between a set of environment states. If the environment is new and *non-predictable*, then the *adaptation plan* followed by the software may not contain guides to adapt the configuration. To face the unknown situation in this case, a new *adaptation plan* can be requested. This new plan will contain the appropriate adaptation actions for changing from/to this new environment. Thus, in this case the labour of the uppermost layer of the architecture helps to face the unknown.

However, not in every case the unknown situation can be solved; maybe because of lack of information about the environment, maybe because of lack of ideal decision methods. In this cases, software is executing in ignorance of its environment. In next subsection it is

described how the software can manage that unavoidable ignorance.

2.2 Self-adaptive systems manage ignorance

Self-adaptive software that changes its configuration in response to unexpected changes in its execution environment should be conscious of the fact that its sensors and prediction algorithms are not perfect, and then it may not have the knowledge at every time about what the real environment and its best configuration are.

To classify different types of lacks of knowledge of self-adaptive software, we describe in the next paragraphs a comparison between the human orders of ignorance and learning processes and the software one. We work upon the five orders of human ignorance proposed in [Arm00], and we reason in which level our self-adaptive system is. The five levels of ignorance in [Arm00] are briefly described as:

- 0th order of ignorance. Lack of ignorance, i.e., knowledge.
- 1st order of ignorance. Lack of knowledge. The subject lacks of knowledge about something but he/she/it is aware of such lack.
- 2nd order of ignorance. Lack of knowledge and lack of awareness. The subject does not know that he/she/it does not know.
- 3rd order of ignorance. Lack of process to find out the lack of awareness. The subject does not have any way to move from not knowing that he/she/it does not know to, at least, be aware of his/her/its ignorance.
- 4th order of ignorance. Meta ignorance. Ignorance about orders of ignorance.

Self-adaptive systems should have knowledge about its execution environment. Nevertheless, since that fact cannot be always ensured, they should at least be able to manage its ignorance. Then, the orders of ignorance can be applied to them. From the information coming from sensors, they create knowledge of their environment and they decide what the execution environment is. In case that the decision regarding the execution environment comes up with a different environment than the currently used, the next step involves either a selection of the adaptation actions -in case that the second layer is ready to cope with the new environment characteristics- or a deliberation about a long-term adaptation plan -in case that the uppermost layer is invoked.

During software execution, the system is in the 0th order of ignorance when its last decision about the execution environment guessed correctly the real environment and the real environment has not changed since then.

During the first moments after an environmental change -e.g., context changes from *a* to *b*-, system stands on the second order of ignorance; i.e., it does not know that it does not know the actual environment because it has not received information from sensors yet. As soon as new information from sensors arrives and the second layer realizes that such information does not fit with the expected values for environment *a*, system changes from 2nd to 1st order of ignorance; i.e., now it does know that it does not know what the current environment

is (*b*). It will remain in that state until it makes a decision about what the new execution environment is. If the decision is successful (it decides that the new environment is *b*), the order of ignorance comes back to 0th again. On the contrary, if it decides erroneously the order of ignorance goes to 2nd.

At least two factors can make the system move to the third order of ignorance: when the change refers to an environmental characteristic that is not measured by any system sensor in the bottom layer, or when the deliberation algorithms in the uppermost layer are not correct.

In the former case, sensors are not monitoring the appropriate environmental properties and then they will continue reporting the same data. Since sensors are the primary exposed mechanism to unveil that something has changed, the system will not know that it does not know the actual environment and it will not have any other mechanism to become aware of such ignorance.

In the latter case, which happens when the uppermost layer has been invoked, the system does its best to unveil the current environment and what properties it can expect from it. To perform this task, it executes the deliberative algorithms that generate the expected execution environments and an adaptation plan between them. However, these algorithms are not adaptive (at least, in this thesis is not considered that such part of code can evolve). Therefore, a wrong algorithm will come up with unsuitable adaptations plans and the adaptation manager in the second layer will be clueless when it has to decide adaptations. Going into the latter case in depth, now the ignorance is not referring to *"system does not know that the actual environment is not the same as its expected environment"* as it happened in the previous cases, but to *"system does not know that its set of expected environments and adaptations between them are not correct"*. However, there can be devised a manner in which the system can belong to the 2nd order of ignorance for this case. This manner is: providing the software with self-evaluating techniques of its expected behavior and comparing the self-evaluation results with the actual ones. If the expected results do not match up with the real ones, the software can activate a warning stating that the last adaptation plan did not work as expected. In this case, another adaptation plan is proactively created by the third layer. Nevertheless, if subsequent self-evaluations show that results do not match for a sequence of plans, the system can infer that it does not know how to generate suitable adaptation plans, and it can raise an alarm notifying humans of that fact. So, having self-evaluating and reporting mechanisms, the system can keep 2nd order of ignorance instead of being on the 3rd one for this case, but it will require human intervention (i.e., to be installed a correct algorithm for the deliberation activity) to eventually move to 0th again.

Probabilistic environment decision The decision about the current environment may not be straightforward. Indeed, the reported values from sensors may fit in more than one execution environment. In this case, starting from sensed information, the generation of knowledge can consist of mathematical treatments of that information. This mathematical treatment can, for example, obtain results as: statistical indicators about the expected values of the environment, their temporal variance, confidence level in the knowledge estimation and confidence level on the adaptation decisions.

Software with probabilistic environment decisions manages its execution from the 1st order of ignorance; i.e., it always knows that it might be executing under an unknown en-

vironment. This software does not stand on ideal 0th order of ignorance but it is also less likely to move to the 2nd order. The reason for not being on the 0th order is that, even if a decision about configuration change is right, the system will always consider the probability of being wrong; i.e., the 1st order of ignorance in the sense *it knows that it cannot be completely sure of its environment*. Regarding the second statement, since it always manages a probability of having decide wrongly, it is obviously aware of its possible ignorance when facing adaptation decisions, which avoids it entering into the 2nd order of ignorance when environment changes. Yet the system can move to the 2nd order of ignorance if it runs incorrect deliberation algorithms and it is provided with self-evaluating mechanisms.

Chapter 3

Self-adaptation for Performance Engineering: Goal Management

This chapter describes our research in the field of Software Performance Engineering (SPE) applied to a recently proposed software paradigm, the *open-world* software [BNG06]. For this type of applications, we investigate the uppermost layer of the reference architecture and the generation of adaptation plans when the extra-functional goal is to improve the application's mean response time.

3.1 Problem description

The open-world software paradigm [BNG06] encompasses and abstracts concepts underlying a wide-range of approaches and technologies; among them, grid computing, publish-subscribe middleware or service oriented architectures. In open-world, an accepted approach considers software as made of *services* provided by *components* elsewhere deployed that interplay without authorities. The software achieves its goals by selecting and adapting services which evolve independently. Then, this software evolves itself in unforeseen manners that depend on third-parties, which means that the performance for this software strongly relies on the performance of the services it trusts. Therefore, current methods to predict "non open-software" performance can now hardly be completely reused in this new context. Consider they make assumptions which now could not take place, for example, to assume service times for software activities as well-known performance input parameters. Now, these activities are executed by third-parties and their execution times can vary in unforeseeable manners. Thus, the evolving behaviors of the third-parties make up the changing execution environment of the client application.

Being self-management an inherent characteristic in open-world software, it is argued that challenges in the former are also present in the latter. Hence, we are convinced that open-world can take advantage of the three-layer reference architecture [KM07, KM09]. At this respect, we want to study if the architecture fits the open-world and if it can bring those

previously enumerated benefits to this context. In particular, this chapter is focussed on how to exploit the architecture for the open-world software to incorporate a performance-aware property. In this regard, the contributions of this chapter are:

- First, we discuss how open-world software could be adapted to the architectural description. In particular, we stress the implications for the architecture to carry out performance-aware reconfigurations in this context. We will accomplish it in Section 3.2.
- Once the architectural implications for performance have been presented, we will address an explanation about the most challenging component in this architecture. The component is an instance of the *adaptation plan generator*, and it is in charge of generating the performance-aware *adaptation plans* or *strategies*. Section 3.3 describes algorithms for this component.
- The last contribution is an example, developed in Section 3.4, that demonstrates the feasibility of the proposed module and shows how the strategies it develops may improve the system performance.

3.2 3-Layer architecture for open-world software

In this section, we describe how to place the reference architecture to the open-world software context and how to manage the performance-aware property. Then, for each layer we have to identify what responsibilities it has to take so the system eventually can accomplish this property. Hence, we are pursuing an architecture for performance-aware open-world software. Figure 3.1 describes an architecture instance for this kind of software.

3.2.1 Component control

As previously discussed, this layer is in contact with the execution environment and has to quickly react to changes produced in it. In the open-world software this means that this layer manages the components making up the current configuration. Therefore, it is responsible for establishing the current bindings and unbindings when a component has to be called.

Concerning performance, we identify for this layer different responsibilities. They are the minimum set an open-world software may need to actually develop activities leading to manage performance aware reconfigurations. Firstly, it will be in charge of tracking the performance of the services involved in the current configuration. Secondly, it has to discover new components offering services equivalent in functionality to those required by the workflow. Finally, it has to be aware about which ones of the current providers are no longer available.

For an open-world software to carry out these responsibilities, *monitor* modules can take charge of all them. These monitors should be incorporated to the target open-world software as a module. For the first task, it will control the time elapsed in the calls to the services, called *providers performance monitors* in Figure 3.1; and for the second and third it will use the normal means in open-world (i.e., through service discovering), called *service discovery*

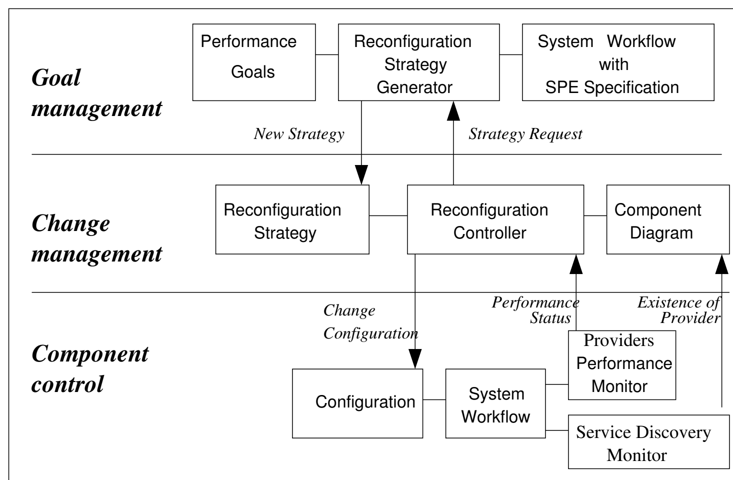


Figure 3.1: 3-layer architecture adapted to performance-aware open-world

monitors. These monitors are the instance of the general *sensors and monitors* in Chapter 2 for performance-aware open-world software.

From a practical point of view, this layer also needs a representation of the *workflow* to be executed and of the set of third-parties that conform the current *configuration*. This part is the instance of the general *software application and infrastructure* in the general architecture in Chapter 2. In this research, we will consider that such workflow has the form of a UML activity diagram while the current configuration will be represented by a UML component diagram (indeed an instance of the one in the Change Management level). Whatever other standard representation could be valid such as BPEL for the first or Darwin component model for the latter.

Status messages from this layer to the upper one are sent in the following moments: *performance status* messages are sent each time the execution of a service ends (informing about the monitored provider's response time); and *existence of provider* messages are sent when it cannot execute the current service in the workflow (e.g., the target component may be unreachable), or when a new provider is discovered. The upper layer can respond with a new configuration.

3.2.2 Change management

The *mission* of an open-world software is obviously carried out through its own execution, here abstracted by the workflow. The workflow execution may need successive self-reconfigurations, that may attend different criteria, for example the cost of the services or the performance. For each criteria of interest, this level can associate at least one *reconfiguration strategy*, which are instances of the generic *adaptation plan* in Figure 2.1. It would also be desirable that a given strategy could gather more than one criteria, for example the previous

two. In any case, for this chapter purposes the interest is that this layer has defined and can manage a performance aware reconfiguration strategy.

For an open-world software to execute strategies, we identify the need of a *reconfiguration controller* module. The inputs for this module would be of course the set of strategies, but also, the *status* messages provided by the monitors and a representation of the environment situation in a UML component diagram (CD). Then, in this approach, the general *adaptation manager* in Chapter 2 comprises the *reconfiguration controller* and also encapsulates its local data in the CD. The output will account for the computed new system configuration. The CD describes for each component its mode, later explained. The current configuration is the subset of currently active components in the CD.

Let us briefly discuss how this layer could manage the components *mode*. The *mode* can be a tuple `<state,MST>`. The first field to be chosen from {`unavailable,standby,active`} and the second to represent the mean service time for the module. Following the proposal in [KM07], the mode could be managed through ports using a `setmode` operation.

Moreover, this layer should `create` the new components and `delete` those no longer useful, remember that the actual `bind` and `unbind` is responsibility of the lower level. Therefore, when monitors in the lower layer report the status, this layer has to manage different situations:

- *A component is no longer available*. *Existence of provider* messages notify of this situation. The reconfiguration controller sets the mode to *unavailable*, and if the component is in use then the controller executes the strategy to find a proper substitute and eventually will report a configuration change.
- *A provider is available for a given service*. Again, *existence of provider* status messages inform of this situation. The information here reported has to include provider's and service's name and the Mean Service Time `t` (MST). As long as this provider has a CD entry, the reconfiguration controller updates it with the new service as `<standby,t>`. Otherwise, it performs a `create` for the provider (as a component) and sets service mode as `<standby,t>`.
- *A service is currently not providing the required QoS*. The report on this situation is given by *performance status* messages. The reconfiguration controller executes the strategy and decides about a service change. If the change is necessary, then it sets the mode of the degraded component from <active,t_1> to <standby,t_1'> and the mode of the one selected from <standby,t_2> to <active,t_2>. When the new configuration is reported, the lower lever takes the responsibility to perform the corresponding `unbind` and `bind`.

Sometimes the current strategy cannot produce a new configuration for the information that monitors have reported (e.g., the selected providers are not available or the performance goal cannot be satisfied). Then this layer will request the *Goal Management* for a new performance aware strategy.

Finally, we remark that the operations in this layer (`create, delete, setmode` and the strategy execution) are supposed to be immediate regarding the system execution time. This is important since this level will not overload the system.

3.2.3 Goal management

From our point of view, the *mission* of the system will be not only to carry out the workflow functionality, but also to do it meeting a *performance goal*. For us this layer has to produce performance aware reconfiguration strategies, then we devise a *reconfiguration strategy generator* module that instantiates the *adaptation plan generator*.

The strategies are sent to the *Change Management* layer and they could be afforded under two assumptions:

- There could exist a library of strategies and the *generator* will decide the appropriate one, for the current request, out of this set.
- The *generator* could actually create the strategy on demand.

In this thesis we just explore the second choice, then the *generator* inputs should be: the *performance goal*, which is the instance of the *extra-functional goals* in the general architecture for QoS; the workflow with a specification of certain performance properties, which plays the role of input *system models* in the general architecture; and the current knowledge about existing providers and their performance, which are the *environment models*. The output is the target strategy that meets the defined *performance goal*. For the sake of simplicity we will consider only system response time. The performance specification will use the MARTE [Obj05] profile.

3.3 Generation of strategies

In this section, we explain the functionality of a *reconfiguration strategy generator* module, which is placed in the *Goal Management* layer. The previous section described the module goal and its interfaces. Algorithms 3.1, 3.2 and 3.3 synthesize the module functionality, i.e., they create a strategy and they report it to the *Change Management* layer. Besides, a warning complements the strategy when it does not meet the performance goal.

Information managed in the algorithms We assume that the system workflow needs to call K external services, s_k, $k \in [1..K]$. In the CD in Figure 3.3, $K = 3$, thought that the same service could be requested in different calls.

A given service s_k may be provided by several components; let L_k be the number of components that provide s_k. We denote as c_{kl}, where $k \in [1..K]$ and $l \in [1..L_k]$, the lth component serving s_k. For example, in Figure 3.3, service s_3 is served by two components c_{31}, c_{32}. Then, $C_k = \bigcup_{l=1}^{L_k} c_{kl}$ is the set of all components that offer s_k, and $C = \bigcup_{k=1}^{K} C_k$ is the set of all components that provide the services specified in the system workflow.

Moreover, we assume that the *environmental models* representation manner is in a Time Table (TT), like Table 3.1, describing for each component its working *phases*. Let J_{kl} be the number of working phases of component c_{kl}; then each phase ph_j, where $j \in [1..J_{kl}]$, is characterized by a pair of real values (S_j^{kl}, SJ_j^{kl}), where S_j^{kl} is the mean service time and SJ_j^{kl} is the mean sojourn time of c_{kl} in ph_j. This timing information would come from the providers or from the system experience monitoring the environment.

A reconfiguration strategy is represented as a directed graph $G = (N, E)$, see example in Figure 3.5. A node $n \in N$ is interpreted as a system configuration, but it is also important to know for each component the *phase* we guess it is working out. An edge $e \in E$ is interpreted either as a change of system configuration, i.e., the component c_{kl_1} that offers service s_k will replace the component c_{kl_2} ($l_1 \neq l_2$) that provides the same service, or as a change in a component phase. For example, in Figure 3.5, the edge from $Node_0$ to $Node_2$ represents a change of system configuration (c_{22} will replace c_{21}), while the edge from $Node_0$ to $Node_1$ models a change of phase in component c_{11} (from ph_1 to ph_2).

An edge is labeled as $\langle s_k, cond\rangle$, where s_k is the service and $cond$ is a ratio representing our minimum *confidence level* for the change to be produced, consider that being stochastic our analyses, then there exist a probability that the strategy fails its prediction (remember the discussion in Chapter 2 arguing that self-adaptive software manages ignorance).

Description of the algorithms Algorithm 3.1 summarizes the strategy generation, it starts creating the strategy initial node (line 2) and from this node produces its adjacent ones (line 11) and the edges that join them (line 16). While there are nodes whose outgoing edges have not been created yet, it keeps creating nodes and edges. Finally, it creates a set of "way back" edges (line 21). Such edges represent either changes of configuration or component-phases due to timeouts instead of a *condition* as it happened to forward edges. The rational behind a "way back" is to bring back the system to a configuration that after some time would be working better than the current one.

Algorithm 3.2 solves the calls in Algorithm 3.1 (lines 2,11), i.e. how to create a node in the strategy. *PhaseList* is a list of pairs $\langle c_{kl}, ph_j\rangle$ that for each $c_{kl} \in C$ assumes its phase $ph_j \equiv (S_j^{kl}, SJ_j^{kl})$. When Algorithm 3.2 creates the initial node, *PhaseList* (line 3) is created assuming that each c_{kl} is in its phase with minimum mean service time. However, for the rest of the nodes (line 5), *PhaseList* is constructed with *ExtractListOfPhases* that will implement an algorithm choosing appropriate phases. In Section 3.4 we will exemplify our proposal for such algorithm. Function *AllPossibleConfigs* (line 9) creates all possible system configurations according to *PhaseList*. Each configuration will parameterize the workflow GSPN that will be evaluated to get the configuration response time (lines 11..13). In particular, the mean service times S_j^{kl} of the components c_{kl} belonging to the configuration, in their current phase ph_j, are used to parameterize the GSPN. As an example, during the creation of the initial node $Node_0$ in Figure 3.5, four candidate configurations are generated (see Table 3.2) and evaluated using the workflow GSPN in Figure 3.4. Finally, the node created by the Algorithm 3.2 (line 15) corresponds to the system configuration with the minimum response time.

Algorithm 3.3 solves the call in Algorithm 3.1 (line 16), i.e. how to create a forwarded edge, not a "way back". Observe that, if service s_k of a given $Node_s$ cannot be replaced by any $Node_t$ with a new *phase* or component, no edge is created (line 1). Function *ExtractListOfPhases*, in Algorithm 3.2, detected this situation and *CreateNode* returned null. When $Node_s$ has an adjacent node $Node_t$, then a direct edge from the source node $Node_s$ to the target node $Node_t$ is created together with its labeling information, i.e., the service s_k and the condition $cond$ (line 5). In particular, $cond$ is a real value computed by the function *SetConfLevel* (line 10), which needs the response time evaluated using the workflow

GSPN for $Node_s$ and $Node_t$ (lines 6-9). A simple example of *SetConfLevel* will be given in Section 3.4.

These algorithms have been implemented in the work in [Fra10].

Algorithm 3.1 Strategy generation

Require: From Goal Management Layer: System Workflow (AD), Performance Goal (PerfGoal)
From Change Management Layer: Components with their timing specification (CD,TT)
Ensure: A New Strategy (and a possible warning meaning that the PerfGoal is not achieved)

```
    {Initialization}
 1: set G = ⟨N, E⟩: N = ∅ {nodes}, E = ∅ {edges}
    {Create Initial Node}
 2: N_0 ← CreateNode(AD,CD,TT,null,null)
 3: set Nodes = ∅
 4: Nodes = Nodes ∪ N_0
 5: while Nodes ≠ ∅ do
 6:   Node_s ← ExtractOneNode(Nodes)
 7:   AlreadyCreated ← CheckNode(N, Node_s)
 8:   if not AlreadyCreated then
 9:     N ← N ∪ Node_s
        {Create Node_s adjacent nodes}
10:     for all k ∈ [1..K] do
11:       Node_t ← CreateNode(AD,CD,TT,k,Node_s)
12:       AlreadyCreated ← CheckNode(N, Node_t)
13:       if not AlreadyCreated then
14:         Nodes ← Nodes ∪ Node_t
15:       end if
          {Create edge from Node_s to Node_t}
16:       Edge ← CreateEdge(Node_s,Node_t,k,TT)
17:       E ← E ∪ Edge
18:     end for
19:   end if
20: end while
21: E = E ∪ CreateWayBackEdges(G,TT)
22: return ¡G, AnalyseStrategy(G,PerfGoal,CD,TT)¿
```

3.4 Example

We exemplify the algorithm of the strategy generation, described in Section 3.3, with an example of a system under development (SUD) that executes three operations, in a sequential manner. All such operations consist in service calls to providers in the open-world environment. The UML system specification is shown in Figures 5.12 and 3.3. The activity diagram

Algorithm 3.2 CreateNode

Require: AD,CD,TT,service (k), current node (node)
Ensure: A node ($\text{conf}_{\text{best}}$)

```
1: set PhaseList = ∅ {vector of vectors}
2: if (k==null ∧ node==null) then
3:    PhaseList ← ExtractInitialListOfPhases(CD,TT)
4: else
5:    PhaseList ← ExtractListOfPhases(CD,TT,node,k)
6: end if
7: set CandidateConfigs = ∅ {set of configurations}
8: set RTs = ∅ {set of configuration response times}
9: CandidateConfigs ← AllPossibleConfigs(PhaseList)
10: for all conf ∈ CandidateConfigs do
11:    GSPN_conf ← CreateGSPN(conf)
12:    rt_conf ← Evaluate(GSPN_conf)
13:    RTs ← RTs ∪ ⟨conf, rt_conf⟩
14: end for
15: conf_best ← FindBestConfig(RTs)
    {The node is a configuration with the min response time:
    ⟨conf_best, rt⟩ ∈ RTs | ∀⟨conf, rt_conf⟩ ∈ RTs : rt ≤ rt_conf }
16: return conf_best
```

(Figure 5.12), annotated with the MARTE profile [Obj05], represents the system workflow (i.e., the *system model*). The type of workload (*GaWorkloadEvent*) is open and requests arrive to the SUD with an exponential inter-arrival time, with a mean of 500 time units (i.e., "tu"). The requests are processed, one at a time, by acquiring (*GaAcqStep*) and releasing (*GaRelStep*) the resource c_0. Each activity step (*PaStep*) models an external service call s_k to a provider in the open-world. In particular, the *extOpDemands* tagged-value is a parameter that is set to the current provider of service s_k and the *extOpCount* tagged-value indicates the number of requests made for each service call.

The component diagram (Figure 3.3) represents the currently available providers of the services required by the system. In particular, component's names are given according to the name of the service they provide. There exists only one provider c_{11} of service s_1, while two providers are available for each service s_2 and s_3. Table 3.1 (TT) shows the working phases, in time units, of the providers. In particular, for each provider c_{kl}, the estimated mean service times S_j^{kl} and mean sojourn times SJ_j^{kl} of the offered service, are given. Both the component diagram and TT make up the *environment models*.

3.4.1 Strategy generation

The Time Table and the UML specification, properly annotated with MARTE, provide the input for the Algorithm 3.1 described in Section 3.3. A parametric GSPN model is then created from the activity diagram (Figure 5.12) that will be used to estimate the mean response

Algorithm 3.3 CreateEdge

Require: source ($Node_s$), target ($Node_t$), service (k), TT
Ensure: The edge between $Node_s$ a $Node_t$ (edge)
1: **if** $Node_t ==$ **null** **then**
2: **return null**
3: **end if**
4: **set** cond = 0.0 {confidence level (float)}
5: **set** edge $= \langle Node_s, Node_t, k, \text{cond}\rangle$
{Computation of $Node_s$ response time}
6: GSPN$_{Node_s}$ ← CreateGSPN($Node_s$)
7: rt$_{Node_s}$ ← Evaluate(GSPN$_{Node_s}$)
{Computation $Node_t$ response time}
8: GSPN$_{Node_t}$ ← CreateGSPN($Node_t$)
9: rt$_{Node_t}$ ← Evaluate(GSPN$_{Node_t}$)
{Computation of the confidence level}
10: cond ← SetConfLevel($Node_s, rt_{Node_s}, Node_t, rt_{Node_t}$,TT)
11: **return** edge

time of the system under different configurations, using the `multisolve` facility of GreatSPN [Gre]. The GSPN model is shown in Figure 3.4 and it is characterized by three rate parameters representing the execution mean rates of the service calls s_1, s_2 and s_3.

Observe that the call to service s_2, in the activity diagram, includes 3 requests (*extOpCount* tagged-value) this is modeled by the free-choice subnet, where the weights assigned to the conflicting transitions *Start_CallS2* and *End_CallS2* are equal, respectively, to $3/4$ and $1/4$.

The first main step of the algorithm (Algorithm 3.1 - line 2), consists of creating the initial node of the reconfiguration strategy graph (Algorithm 3.2). This is accomplished by assuming that each provider works under the best mode. We consider, then, the minimum estimated (mean) service times from each provider, i.e., $S_1^{11} = 5tu$, $S_1^{21} = 10tu$, $S_1^{22} = 35tu$, $S_1^{31} = 20tu$ and $S_1^{32} = 30tu$. There are four possible system configurations: for each one, we instantiate the parametric GSPN, in Figure 3.4, by setting the rate parameters $\lambda_{S1provider}, \lambda_{S2provider}$ and $\lambda_{S3provider}$ to the inverse of the considered service times $S_1^{kl}(k = 1, 2, 3)$ of each current provider of services s_1, s_2 and s_3, respectively. Once instantiated, the GSPNs are solved and the system (mean) response times are computed (see Table 3.2).

In the strategy graph (Figure 3.5), the initial node $Node_0$ corresponds to the configuration that revealed the minimum system (mean) response time. Observe that, in this simple example, active providers in the initial configuration correspond to those ones having the minimum service times. However, this property does not always hold in a general case where several providers contend for shared resources.

In the next main step of the Algorithm 3.1 (line 11), the nodes adjacent to the initial one are created, considering that the active providers in $Node_0$ can degrade their performance. Eventually, there will be three configuration nodes adjacent to the initial node, one for each

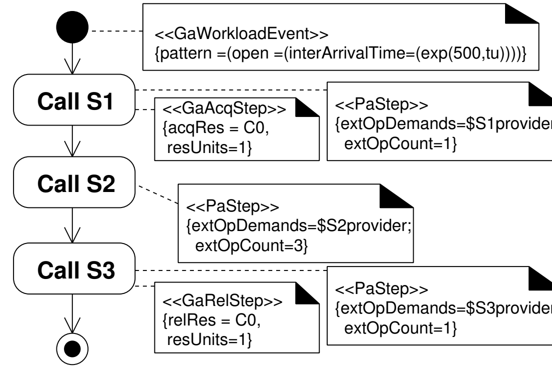


Figure 3.2: UML activity diagram

external service requested by the SUD (Figure 3.5). Let us consider the creation of the first two nodes $Node_1$ and $Node_2$ adjacent to $Node_0$: the algorithm will iterate over the created nodes to produce their adjacents, until all the possible system configurations are examined.

$Node_1$ is added considering that the active provider of service s_1 in $Node_0$ (i.e., c_{11}) changes its *phase* from ph_1 to ph_2, i.e., c_{11} is answering to service requests with a mean service time of $20tu$, instead of $5tu$. Since c_{11} is the unique provider of s_1, the $Node_1$ is characterized by the same active providers as $Node_0$ as well as the same provider mean service times but the one of c_{11}, which is equal to $20tu$. The GSPN model in Figure 3.4 is used to compute the system mean response time of the configuration $Node_1$.

$Node_2$ is created assuming that the active provider of s_2 in $Node_0$ (i.e., c_{21}) changes its *phase* by increasing the mean service time from $10tu$ to $70tu$. Then, four candidate configurations were possible: two of them still include c_{21} as active provider of s_2 with degraded performance. They correspond to the first and the third configuration in Table 3.2 with the provider c_{21} in phase ph_2. In the other two configurations, the active provider of s_2 is c_{22} (i.e, the second and the fourth configuration in Table 3.2). The GSPN model in Figure 3.4 is then used to select the best configuration among the candidates, that is the one with the minimum system (mean) response time. Then, the $Node_2$ actually corresponds to the configuration with the minimum system (mean) response time, i.e., $177.6tu$.

Once a new adjacent node is created, the algorithm generates the corresponding forward edge (Algorithm 3.1- line 16). An edge from $Node_s$ to $Node_t$ includes information about the service s_k and the goodness of the prediction (confidence-level) for the *reconfiguration controller* to decide whether it is worth to change the configuration from $Node_s$ to $Node_t$. Observe that, since we are dealing with the open-world environment, every decision about the providers is based on predictions. We propose an ad-hoc heuristic that works under the open workload assumption and considers the performance goal (i.e., obtain the best system mean response time) as well as the available timing specifications (i.e., provider working phases).

Let us consider an edge from $Node_s$ to $Node_t$ where the source and the target nodes have different active components, such as $Node_0$ and $Node_2$ in Figure 3.5. The computation of

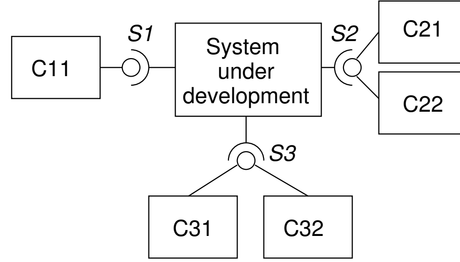


Figure 3.3: UML component diagram

the corresponding minimum confidence level is related to two quantities:

- The performance improvement when the system reconfigures properly, that is the provider has changed its phase and the strategy realizes it (e.g., the provider c_{21} has changed from ph_1 to ph_2 and the system moves from $Node_0$ to $Node_2$). This is estimated as:
$$Perf_{improve} = rt_{s|c_{kl} \leftarrow ph_{j+1}} - rt_t,$$
where $rt_{s|c_{kl} \leftarrow ph_{j+1}}$ is the system mean response time with the same active providers as in $Node_s$, but changing the working phase of provider c_{kl} from ph_j to ph_{j+1}, and rt_t is the system mean response time in $Node_t$.

- The performance loss when the system reconfigures due to a wrong prediction, that is the provider has occasionally had a slow execution, but it has not really changed its current phase, however the system moves to the target node. This is estimated as:
$$Perf_{loss} = rt_t - rt_s,$$
where rt_s is the system mean response time in $Node_s$.

Then, the minimum confidence level is given by the formula:

$$conf_level = \frac{Perf_{improve}}{Perf_{improve} + Perf_{loss}}. \tag{3.1}$$

When the source and target nodes of an edge have the same active components, such as $Node_0$ and $Node_1$, the minimum confidence level is computed as $conf_level = \frac{rt_s}{rt_t}$.

Finally, the *way-back* edges are created (Algorithm 3.1 - line 21) to allow the system to move back to a previously considered configuration after a (mean) sojourn time period in the source node. So there will be an edge from $Node_s$ to $Node_t$, labeled with a mean sojourn time period as a timeout, if there exists a provider c_{kl} in $Node_s$ with its final phase $ph_{J_{kl}}$ and in $Node_t$ with its initial phase ph_1. In Figure 3.5, way-back edges are dashed and, for readability, only five of them are shown. The choice of the ideal mean sojourn time period that allows the system to achieve the performance goal (i.e., minimum response time) is a

Provider working phases (in time units, i.e., tu)			
	ph_1	ph_2	ph_3
C11	(5,3000)	(20,6000)	
C21	(10,6000)	(70, 2000)	(250,2000)
C22	(35,6000)	(140,4000)	
C31	(20,2000)	(70,2000)	
C32	(30,∞)		

In format $ph_j = (S_j^{kl}, SJ_j^{kl})$

Table 3.1: Time Table of open-world providers (TT)

future work issue. In the example, we set such period equal to the mean inter-arrival time of a service request to the SUD (i.e., $500tu$).

In order to validate our proposal, we carried out the analysis of the system, considering several assumptions: the system does not follow the strategy modeled by the reconfiguration graph in Figure 3.5 (case 1), and the system undergoes reconfigurations according to the strategy graph (case 2). We obtained the following results for the system mean response time: $494tu$ (case 1) and $436tu$ (case 2). This means that partially applying our performance aware reconfiguration (eight nodes in Figure 3.5) we have improved the system response time in 11%.

3.5 Conclusion

Along this chapter, we have learnt that there exist a lot of challenges for the performance prediction of the open-world software to become a reality. However, we believe that this chapter has proposed a clear architecture for performance-aware open-world software, which means an attempt to comprehensively accomplish most of such challenges. From this architecture, we have explored the uppermost layer and how to generate reconfiguration strategies, that can reconfigure a system while its performance goal has to be achieved. Our *generation* technique tried to show up where the problems are and it demonstrates a possible solution using Petri nets. However, other generation approaches could be feasible and would be desirable. We validated our solution through an example. A direction to do research in the future work is to include more extra-functional requirements into the strategy generator functionality, such as dependability. As a technical detail, in this work we have not considered network transmission delays, which could be significant in some service-oriented applications that operate on the Internet; however, these delays can be easily incorporated through the UML deployment diagram.

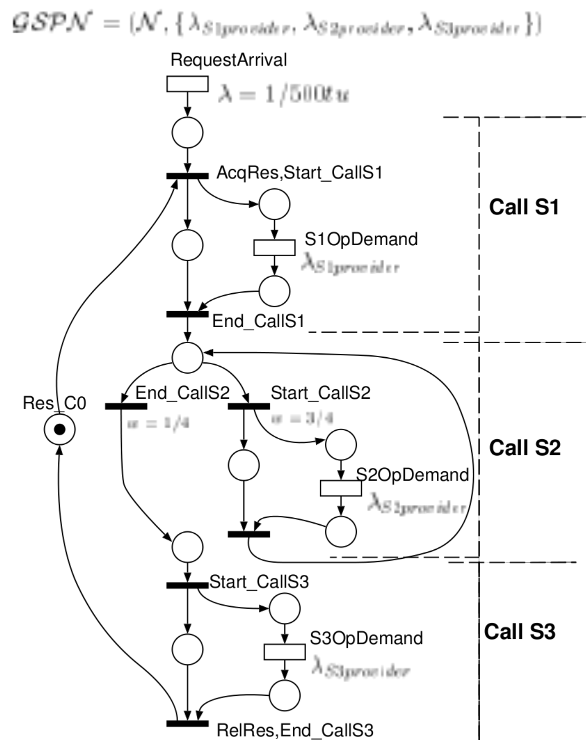


Figure 3.4: Parametric GSPN

3.6 Related work

We believe that the idea of introducing a reference architecture coming from self-managed systems in the open-world software is original. Therefore, the presented solution to introduce and manage performance aspects in such architecture is also new. Probably, the closest work is the one in [GT09], where authors also evaluate performance in open-world assuming components that can evolve independently and unpredictably. However, they use queueing networks and further comparisons are difficult since they address other challenges in the open-world instead of the strategy generation problem.

Although not focussed on the open-world paradigm, Menascé [SM05, MD07, MRG07] evaluates service-based software. These works use brokers to negotiate and manage QoS parameters that are well-known and reliable. Our approach, that at this respect was inspired in [BGG04, LR04], is completely different since it tracks open-services to predict current QoS. This means that the quality of our predictions have to be of inferior quality, but consider that being open our environment, we have to deal with untrusted third-parties. Also in

Mean response time estimation (in time units, i.e., tu)	
C11:ph_1 C21:ph_1 C31:ph_1	60.5
C11:ph_1 C22:ph_1 C31:ph_1	177.6
C11:ph_1 C21:ph_1 C32:ph_1	72.5
C11:ph_1 C22:ph_1 C32:ph_1	193.8

Table 3.2: System components candidates

[PSL03] is addressed the problem of guaranteeing the QoS of untrusted third-party services. They propose a framework to choose services offering best QoS, in this work the workload is balanced among several providers to support some kind of fault tolerance.

The work of Garlan in [GS02] also proposes an architecture for performance evaluation but restricted to self-healing systems, besides they do not use formal methods. Our work has also been inspired by the work in [OMT98], which proposes an architecture to manage the adaptation for evolvable systems, although that work does not deal with performance evaluation.

Work in [CMI07] deals with adaptation management and proposes a Performance Management Framework. In this framework, alternative configurations are dynamically created by evaluating reconfiguration policies and monitored data, rather than store from the beginning a predefined set of possible configurations as we do.

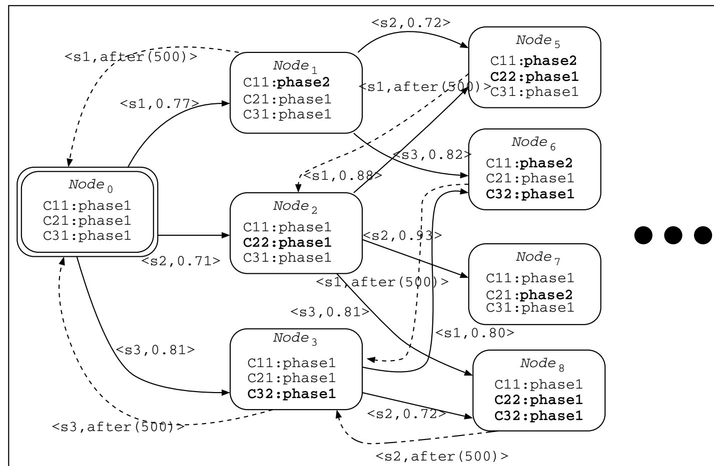


Figure 3.5: Partial reconfiguration strategy graph

Chapter 4

Self-adaptation for Performance Engineering: Change Management

In this chapter we investigate adaptation decisions for SOA applications that are built on services offered by third party providers. A proper decision about when a provider should be substituted can dramatically improve the performance of the application. We propose hidden Markov models (HMM) to help service integrators to foretell the current performance state of third-parties. The HMM manipulation is done by the Change Management layer of our reference architecture.

Next sections emphasize the decision task of a performance-aware self-adaptive software and we propose probabilistic methods that allow the software to decide when and how its adaptation actions should be executed.

4.1 Problem description

Service Oriented Architectures (SOA) provide abstraction mechanisms to ease building complex, heterogeneous and distributed software systems. A pillar of these architectures is the concept of *service*, which is a software entity that allows to execute functionalities in a loosely coupled manner and whose interface is well-described. As in [CCG+09] and [CDPEV08], we call *abstract service* to the offered functionality, and *concrete service* to the particular service the provider exposes. A service can be offered by several providers and they are differentiated by their QoS. Moreover, although the QoS of providers were similar in the long term for the same service, in a given moment providers are exhibiting different QoS; e.g., their performance can differ due to the current workload.

Concrete services can be invoked as part of a complex service oriented system which acts as a *service integrator*. A service integrator can follow a workflow of requests to services provided by third-parties. Working with third-parties adds new concerns since services are

deployed and maintained without control of our organization, which implies that their QoS variations are unpredictable. Furthermore, new services can be developed or existing ones can be retired from the market.

This chapter deals with the tasks of deciding *when* and *how* adaptation actions should be executed to maintain the system working with a suitable performance. Using adaptation, the software can change, for example, some of the services it uses or its overall service composition [CCG+09, CGK+11]. Here, adaptation actions carry out a change in the service provider that will be requested to execute the required abstract service. The approach relies on some knowledge of the performance of service providers. Concretely it assumes that the number of different "performance states" for a given provider is finite, so it supports providers acting with varying performance. Moreover, it is also considered the mean sojourn time that providers spend in each performance state and the probability of moving from one state to another. Since services are provided by third-parties, service integrators cannot assume knowledge of the actual performance of the third-parties. On the contrary, integrators should collect data during runtime to predict this information. Later, integrators will use such prediction to decide whether adapt the request of the system to providers with higher performance.

To foretell the performance states of providers based on monitored information related to their response time we propose the utilization of hidden Markov models (HMMs) [Rab89]. They are described in Section 4.2, where we specify the type of HMMs we use and how they are useful for our purposes. After, Sections 4.3 and 4.4 develop the approach. Finally, Sections 4.5 and 4.6 explain our conclusions of this chapter and the related works, respectively.

4.2 A formal model for SOA providers

4.2.1 Hidden Markov models

A hidden Markov model (HMM) [Rab89] is a double stochastic process, where one of the processes is observable and unobservable the other. We use this model to predict the state of the unobservable process by means of the observable one. An HMM is characterized by the following [Rab89]:

1. The number of states in the model, N. These states are hidden, and they use to represent the meaning of what is intended to be predicted. Individual states are denoted as $S = \{s_1, s_2, ..., s_N\}$, among them the actual state at time t is denoted as q_t.

2. A state transition probability distribution matrix $A = \{a_{ij}\}$, $1 \leq i, j \leq N$. Transition probabilities are assumed normalized, i.e., $\forall i \sum_{j=1}^{N} a_{ij} = 1$.

3. An initial state distribution $\pi \in (\mathbb{R}^+ \cup 0)^N$, where $\pi(i) = P[q_1 = s_i]$.

The former three characteristics conform to a discrete time Markov chain (DTMC). The state of the DTMC at each instant of time will be the unobservable process.

4. Number of distinct observation symbols per state, M. They are denoted as $V = \{v_1, v_2, ..., v_M\}$.

5. The observation symbol probability distribution in state i, $B = \{b_i(k)\}$, where $b_i(k) = P[v_k\ at\ t|\ q_t = s_i]$.

Continuous observation density HMMs Since our observations will be measured times, which are continuous values, the number of observed symbols $M \rightarrow \infty$. Moreover, such measures are expected to follow an exponential distribution with parameter λ_i. Therefore, we consider the HMM to be a single continuous observation, then items 4 and 5 change to: probability distribution function of observation O in state i is $b_i(O) = \lambda_i e^{-\lambda_i O}$. [1]

Continuous time HMMs Advancing descriptions in next subsection, our approach models the non-observable behavior of third-party providers. Such behavior is defined by several states, mean sojourn time in each state, and transition probabilities among states. So, it will be needed to model duration of states in the HMM. Unfortunately, citing [Rab89], *perhaps the major weakness of conventional HMMs is the modeling of state duration*. To overcome this weakness, we use the modeling approach in [WWT02], and we will work with continuous time Markov chains (CTMC). As a result, items 1,2 and 3 will describe a CTMC. Items 1 and 3 do not change their meaning, but transition probabilities are converted into transition rates taking into account both the mean sojourn time in each state (Soj_i) and the probability to change from state s_i to state s_j ($p_{s_i s_j}$) where ($\sum_{s_j} p_{s_i s_j} = 1 \wedge p_{s_i s_i} = 0$). Consequently, item 2 is redefined: There is a state transition rate matrix $R = \{r_{ij}\}, 1 \leq i, j \leq N$, where $r_{ii} = \frac{-1}{Soj_i} \wedge \forall i \neq j, r_{ij} = \frac{p_{s_i s_j}}{Soj_{s_i}}$

From now on, we call CT-HMM to an HMM with continuous observation density, and its state change behavior is given by a CTMC. Figure 4.1 depicts an example of CT-HMM.

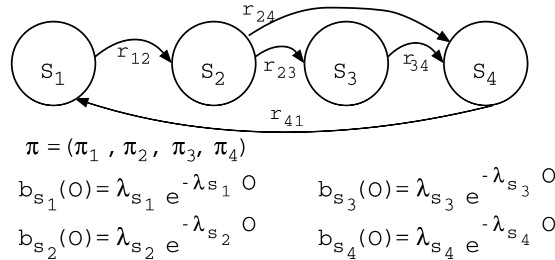


Figure 4.1: CT-HMM example

Note that the word “hidden” in hidden Markov models is not referred to the ignorance of model parameters. On the contrary, it is referred to the actual sequence of states through which the model has passed to generate the sequence of observations; i.e., to the unobservable

[1] $b_i(O)$ can be any finite mixture of log-concave or elliptically symmetric densities $\mathfrak{N}$ [Rab89], $b_i(O) = \sum_{m=1}^{M} c_{im} \mathfrak{N}(O, \mu_{im}, U_{im})$ where c_{im} are mixture coefficients, μ_{im} are means and U_{im} are covariance matrices. In our case, the mixture has only one component, which is the exponential probability distribution function.

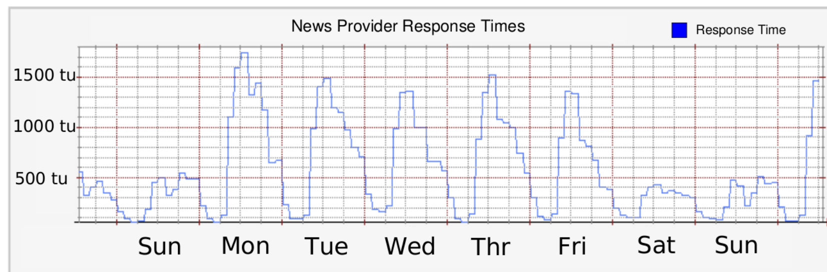


Figure 4.2: Response times of a service provider

process. Thus, these models continue being "hidden" even if all of their parameters are known. This characteristic is explicitly noted in works of different research areas such as [Sch06b, TC11, ZKSZ12]. Moreover, also a section in [Edd04] nicely explains what the hidden part is through an example in the biotechnology research area.

4.2.2 HMM representation of SOA providers

Let us consider the case of a client with an *abstract* service that requests for an item of news, and there exist several providers of news that offer the *concrete* service. The information the client can manage, about the behavior of the providers, is the one it can acquire monitoring their performance. Let us also assume that Figure 4.2 shows an example of the monitored response time of a provider. In that figure we can observe that: 1) during the night, the provider shows a low response time, which is around 100 time units (tu), maybe since few clients (human or service integrators) are consulting news; 2) during the weekend, less than 500tu are needed to generate the response; 3) during daylight of working days the response time increases up to 1100tu, maybe because the workload to request news increases; 4) in such daylight, there can also be peak zones where the time to receive a response suddenly reaches maximums. Peak zones can be predictable or unpredictable. For example, a predictable one is at the beginning of the day (around 9:00 hours), when lots of people may want to read the news; while an unpredictable one may occur when an important event happens (e.g., a crime report). From the information in Figure 4.2, the client could group the behavior of the provider in four states: *daylight, daylight-peak, night, weekend*. See that another provider of the same news service can show different response times (better in certain moments and worse in others). The variation can be due to the workload the provider supports in each moment, which indeed is an unknown information from the client view. Moreover, the long-term behavior could even be the same (same states and mean expected response time), but they are showing different response time in a given moment just because they belong to different time-zones.

The client should address each service call to the provider that shows best performance at the moment the request is delivered. Then, the client has to predict the future expected response time of the providers based on the response times it has already monitored. Besides,

providers are not monitored strictly periodically but when their services are requested. This fact can make prediction methods that are exclusively based on monitored response times lose accuracy. Since we pursue accurate prediction methods, we consider two concepts to carry out providers performance prediction: monitored response times and the time instant they are measured. CT-HMMs provide mechanisms to represent both concepts. Then, we propose them for the client to represent the behavior of a provider:

- A continuous time Markov chain (CTMC) where each state represents a state of the provider (*night, daylight,...*) and the client knows their mean sojourn times s_s (Soj_{s_s}) and the probability to change from a source state to a target state ($p_{s_s s_t}$ and $\sum_{s_t} p_{s_s s_t} = 1$).
- Probability of the value of observations in each state. In this case, observations are the response times monitored from service calls. For us, the expected response time follows an exponential distribution in each state. Therefore, the probabilities of observations in state $s_i \in \{night, daylight, ...\}$ are $b_{s_i}(O) = \lambda_{s_i} e^{-\lambda_{s_i} O}$, where λ_{s_i} is the inverse of the mean expected response time in state s_i and O is the observed time by means of monitoring the provider.
- An initial state distribution π_0. Since no knowledge about the state of the provider is known when the service integrator starts its execution, we assume the initial state distribution to be the steady-state solution of the CTMC $\pi_0 = \pi_{steady}$.

Note that in this model, states are hidden but transition rates among states are known, as well as the expected mean response time in each state. CT-HMM manages two time parameters when receives each observation k: the monitored response time (O_k) and the time instant when such observation has been received (t_k). That is, the service integrator stores the absolute time in which each response has been received, being $t_0 = 0$ the instant where the integrator was switched on.

4.3 State prediction and configuration adaptation

As discussed in previous section, the behavior of providers can be formally represented and the client, or service integrator, can use measured response times to foresee in which state a provider should be currently executing. Besides, service integrators need abilities to change the system configuration, i.e. to select for the current request the provider that has been predicted to be in the state with best response time.

4.3.1 Prediction of the provider state

The CT-HMM proposed in Section 4.2.2 will be useful to predict for a provider the probability distribution of its states. Concretely, using the CT-HMM of provider p, we can predict the state probability (π_k^p) when observation O_k^p is received at time t_k^p considering: the calculated state probability of the previous observation (π_{k-1}^p), the observed response time O_k^p and the amount of time elapsed, $t_k^p - t_{k-1}^p$, since the previous service call to p:

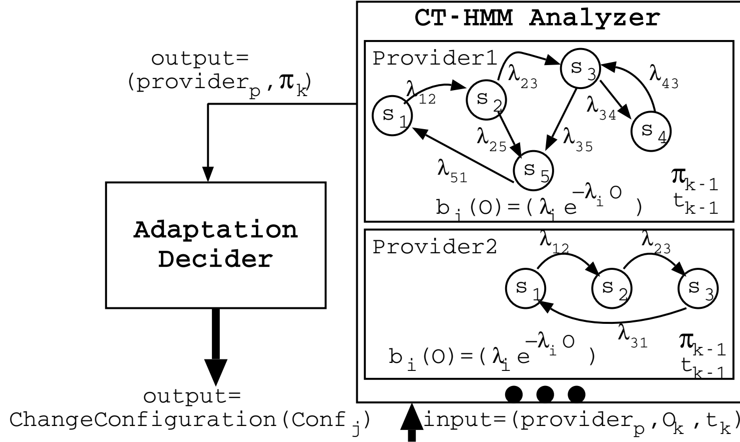


Figure 4.3: Modules for configuration changes

$$\pi_k^p(i) = CalcTransientProb(\pi_{k-1}^p, s_i, t_k^p - t_{k-1}^p) \cdot b_i^p(O_k) \cdot c \tag{4.1}$$

being $b_i^p(O_k) = \lambda_i e^{-\lambda_i O_k}$, i.e., the probability of provider p to receive an observation with value O_k being in state i, and being c a constant to normalize the vector to $\sum_i \pi_k^p(i) = 1$. As previously mentioned, before the first observation (O_1^p), the state probabilities correspond to the steady-state distribution ($\pi_0^p = \pi_{steady}^p$).

To operate with the CT-HMM, which considers CTMCs, *Calc-TransientProb* uses CTMC transient analysis equations to calculate the probability of being in state s_i after $t_k^p - t_{k-1}^p$ time units, being the π_{k-1}^p state probability distribution at time t_{k-1}^p.

Figure 4.3 (right hand side) depicts a supposed software module, we call *CT-HMM Analyzer*, aimed at predicting for a provider the probability distribution of its states. The module stores for each provider the corresponding CT-HMM. When the system advertises that a service call has finished, this module receives as input information, the name of the provider ($provider_p$), the monitored response time (O_k), and the current time (t_k); then it computes and stores π_k^p. When the system requests for the current π_k^p, the expected probability distribution of the states of a provider, this module computes it using: π_{k-1}^p (that calculated when the last observation of p was received) and t_{k-1}^p (the moment when the observation was received).

4.3.2 Adaptations based on state predictions

The probability distribution of the states of the providers, π_k, enables service integrators to select providers offering best response time. Figure 4.3 (left-hand side) depicts a supposed software module, we call *Adaptation Decider*, aiming at deciding system configuration changes.

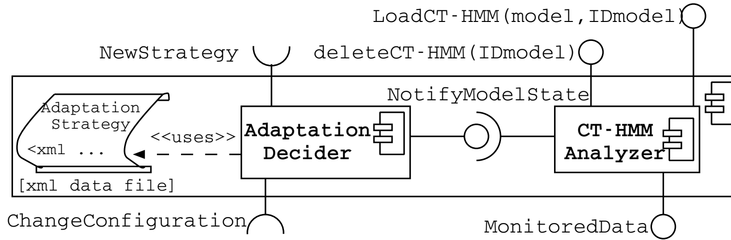


Figure 4.4: Abstract view of Figure 4.3.

We mean by "configuration change" the replacement of a service provider by another one, indeed offering best response time.

Being the decisions of the *Adaptation Decider* based on predictions, there can happen fails or hits. Fails occur when: 1) the provider has actually changed its state but the prediction does not advise it, we call it "false negative"; 2) the provider has not changed its state but the prediction erroneously advises a change, we call it "false positive". Likewise, a decision hit happens when: 1) no change is advised when it was not needed (called "no-adaptation hit"); 2) or a configuration change is advised when needed (called "adaptation hit"). From the point of view of a self-adaptive software system "false negatives" are lost opportunities to improve system performance, see that a non-adaptive system will always miss such opportunities. However, "false positives" can make the self-adaptive system work worse than a non-adaptive one.

The *Adaptation Decider* calculates the system configuration that is expected to show the lowest mean response time for the system workflow execution. Algorithm 4.1 describes such calculation: for each possible configuration (line 4) calculates the weighted mean of the mean response time for each state combination of a provider (lines 6-10); the weight of each term is the probability for providers to be in the state expected in that state combination (line 8). Finally, it is selected the configuration that offers the lowest weighted mean response time.

Algorithm 4.1 has some combinatorial executions (calculation of mrt_i and $Conf_i$) that make it not practicable. The remainder of the subsection discusses three techniques that improve it, we discuss the improvement they achieve and the kind of prediction error they incur.

Most probable state: This technique will change system configuration taking into account for each provider its current most probable state. It will pick the state with lowest response time and consequently selects the provider this state belongs to.

This technique will incur in a large amount of false positives. See for example Figure 4.5: say the workflow consists of only one service call to $S1$. There are two providers (P_{11} and P_{12}), so the system can execute in two configurations ($Conf1$ and $Conf2$). Each provider can execute in 3 states, s_1, s_2 and s_3, with different mean response times as shown in Figure 4.5. Initially, the state probability distributions could be: $\pi^{P_{11}}(1) = 0.3, \pi^{P_{11}}(2) = 0.37, \pi^{P_{11}}(3) = 0.33$ and $\pi^{P_{12}}(1) = 0.15, \pi^{P_{12}}(2) = 0.25,$
$\pi^{P_{12}}(3) = 0.6$.

Algorithm 4.1 Algorithm of the *Adaptation Decider*

Require: $AbstractServices(AS), ConcreteProviders(CP)$, $ProvidersStateDistributions$ (π_k)
Ensure: Conf, the configuration with the lowest expected MRT.
1: **set** $Conf_i$ {Configuration, selection of a CP for each AS}
2: **set** $StateComb_{ij}$ {Possible combination of providers states in a Configuration $Conf_i$}
3: **set** $mrt_i = 0.0$ {weighted mean system response time in configuration $Conf_i$ being its providers state distribution π}
4: **for all** $Conf_i \in (AS, CP)$ **do**
5: $mrt_i = 0.0$
6: **for all** $StateComb_{ij} \in Conf_i$ **do**
7: $mrt_{ij} \leftarrow CalculateMRT(StateComb_{ij})$
8: $probState_j \leftarrow CalculateProbability(StateComb_{ij}, \pi_k)$
9: $mrt_i \leftarrow mrt_i + mrt_{ij} \cdot probState_j$
10: **end for**
11: **end for**
12: **return** $Conf_i \mid \forall mrt_{i'}, mrt_i \leq mrt_{i'}$

Let us assume the system in $Conf_1$, since the most probable state for P_{11} is s_2, the expected mean response time is 50 tu. Now, let us assume that the *CT-HMM analyzer* calculates a new state distribution for P_{11}: $\pi^{P_{11}}(1) = 0.2, \pi^{P_{11}}(2) = 0.39, \pi^{P_{11}}(3) = 0.41$. So, now the most probable state for P_{11} is s_3 and the system mean response time is 140 tu (see Figure 4.5). In addition, the most probable state for P_{12} is s_3, whose expected mean response time is 110tu. Since $110 < 140$, the decision will be to change from $Conf_1$ to $Conf_2$. This decision has a very high probability to be a false positive, because if all state probability distributions had been taken into account, the expected mean response time in $Conf_1$ would have been less than in $Conf_2$, and no reconfiguration would have been proposed (then being a no-adaptation hit). Indeed, applying Algorithm 4.1, the result would have been: $(0.2 \cdot 10 + 0.39 \cdot 50 + 0.41 \cdot 140) < (0.15 \cdot 20 + 0.25 \cdot 80 + 0.6 \cdot 110)$ which indicates that it is better to remain in $Conf_1$.

This technique is faster than Algorithm 4.1 since it only looks for one state for each provider (the most probable) and executes a comparison between pre-calculated mean response times. On the other hand, it needs to have pre-calculated and stored the expected mean response times for each provider configuration, which can be costly for large service based systems.

Most probable state with "sureness": This technique still considers the most probable state for each provider i (mps_i), but it also takes into account its probability and the expected improvement in the system response time. The technique calculates a "sureness" value (*Sr*), based on the response time of the system considering source and target configurations ($Conf_s$ and $Conf_t$), as $Sr = \frac{MRT(Conf_t(mps_i))}{MRT(Conf_s(mps_j))}$. Moreover, it calculates a probability $PgoodPred = \pi_i(mps_i) \cdot \pi_j(mps_j)$. The system will change configuration only if $PgoodPred \geq Sr$. Note that when $PgoodPred$ is high -almost one-, the system will

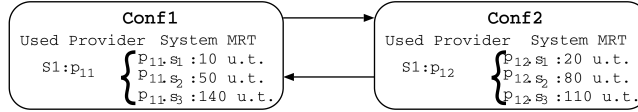


Figure 4.5: Example of two system configurations

reconfigure even when the performance in $Conf_t$ is not very much higher than in $Conf_s$.

This technique executes as fast as the previous one since it also only needs to look for the most probable state probabilities. However, this technique avoids some adaptations that would most likely incur in a false positive. For example, in Figure 4.5, $Sr = \frac{110}{140} = 0.7857$ and $PgoodPred = 0.41 \cdot 0.6 = 0.246$, then $PgoodPred \ngeq Sr$ and the system would not incur in a false positive adaptation, as it happened in the previous one. The technique also avoids false positives that are due to "not as much sure of the most probable state probability as to reconfigure". However, since it only takes into account the probability of the most probable state, it still incurs in false positives related to "in which states are the probabilities that are not in the most probable state of $Conf_s$ providers".

Fail compensation: This technique not only takes into account the most probable state, but the whole state distribution. It reduces the complexity of inner loop in Algorithm 4.1 because (see lines 6-9) it does not calculate mrt_s for each possible combination of states $StateComb_{sj}$ in a source configuration $Conf_s$, however it considers a pre-calculated mean. Concretely, this mean value is pre-calculated for each state of a provider $p_i s_j$ in a configuration, and it represents the mean response time of the system when p_i is in state s_j and considers the steady state distribution for the rest of the providers in $Conf_s$. Therefore, this value represents the mean of the mean response times $mmrt$. Following this technique, the number of loops is $N_{p_i} \cdot N_{p_k}$ instead of $\prod_{p_i \in providers} N_{p_i}$ where N_{p_i} is the number of states of provider p_i.

The technique will cause false positives due to the use of steady state distributions to pre-calculate $mmrt$, whereas in the complete loop in Algorithm 4.1 the actual state probabilities distributions are considered. To mitigate them, it does not reconfigure just when the expected response time in $Conf_t$ is lower than in $Conf_s$, it also considers the "expected profit" when the decision is a hit or the "performance loss" when the decision is a fail (false positive or negative). Then the adaptation is carried out when $ProfitWhenHit > LossWhenWrong$. $ProfitWhenHit$ is calculated as:

$$\sum_{s_j \in p_i} (\pi_{p_i}(s_j) \cdot \sum_{s_l \in p_k} \pi_{p_k}(s_l) \cdot coeff_{prof}(p_i s_j, p_k s_l))$$

where $coeff_{prof}(p_i s_j, p_k s_l)$ is the function

$$coeff_{prof}(p_i s_j, p_k s_l) = \begin{cases} \frac{mmrt(p_i s_j)}{mmrt(p_k s_l)} & \text{if} \quad \frac{mmrt(p_i s_j)}{mmrt(p_k s_l)} > 1 \\ 0 & otherwise \end{cases}$$

To calculate $LossWhenWrong$ it is also used the previous formula but changing the coefficient for $coeff_{loss}(p_i s_j, p_k s_l)$ where

$$coeff_{loss}(p_i s_j, p_k s_l) = \begin{cases} \frac{mmrt(p_k s_l)}{mmrt(p_i s_j)} & \text{if} \quad \frac{mmrt(p_k s_l)}{mmrt(p_i s_j)} > 1 \\ 0 & otherwise \end{cases}$$

4.4 Integrating the adaptive configurations into an architectural solution

In this section we integrate the theory previously described for software that operates in the open-world into the architecture for performance-aware self-adaptive software.

Benefits of integrating our proposal into this architecture are clear: we are approaching to a complete reference architecture for open-world software, that can meet performance requirements while manages uncertainties in the environment through a formal model.

Figure 4.4 appears now embedded within the shadow part of Figure 4.6, which clearly describes how the new proposal fits in the 3-layer architecture. The proposal in this chapter executes the task of the general *Adaptation Manager* of the architecture in Figure 2.1. Both the *Adaptation Decider* and *CT-HMM Analyzer*, are in the *Adaptation Manager* scope. Moreover, to ease the applicability of the architectural approach in a non-HMM based solution, just for generality, we refer to the *CT-HMM Analyzer* using its functionality name: *Providers Performance Predictor*.

Regarding the *Adaptation Manager*, firstly, generates input values of the *Providers Performance Predictor*. In Figure 4.3, they are $\texttt{provider}_p$, $\texttt{O}_k$, and $\texttt{t}_k$, which now respectively match with information about *Who, responseTime* and *When* in Figure 4.6. The generation of this input might need a syntax translation depending on the language of the received *status* messages from the lower layer. Although this technicality about fitting interfaces has not been completely addressed in this work, we do not disregard it and we consider that a translation may exist through a conversion of input *status* messages to *ResponseTimes, Who,* and *When*. Configuration changes are decided using this information, then updating the internal model that stores which one is the main provider for each service and producing a *ChangeConfiguration* output. This output is forwarded in *Change Actions* message to the component control layer.

Regarding the integration of the CT-HMMs management in the architecture some issues need to be clarified. The management of these models is divided into two entities: the entity that creates and parameterizes the models, called *Provider Performance Behavior Analyzer*; and the entity that uses them to predict their current state, the *Providers Performance Predictor*.

The functionality of the former is not achieved by a simple operation but requires complex computations. Besides, its operation is not called frequently. Therefore, it is reasonable to place this entity in the uppermost layer. This functionality has not been addressed in this work. However, this is not a neglected part. Indeed, in the hidden Markov models theory, this is one of the typical studied problems, which means to find out the most probable parameters of an HMM from an observation sequence.

The latter entity was presented in Section 4.3 without taking into consideration some challenges in open world. Then, it now should offer additional interfaces: *loadCT-HMM(model,IDmodel)* and *deleteCT-HMM(IDmodel)*. *Adaptation Manager* will use these

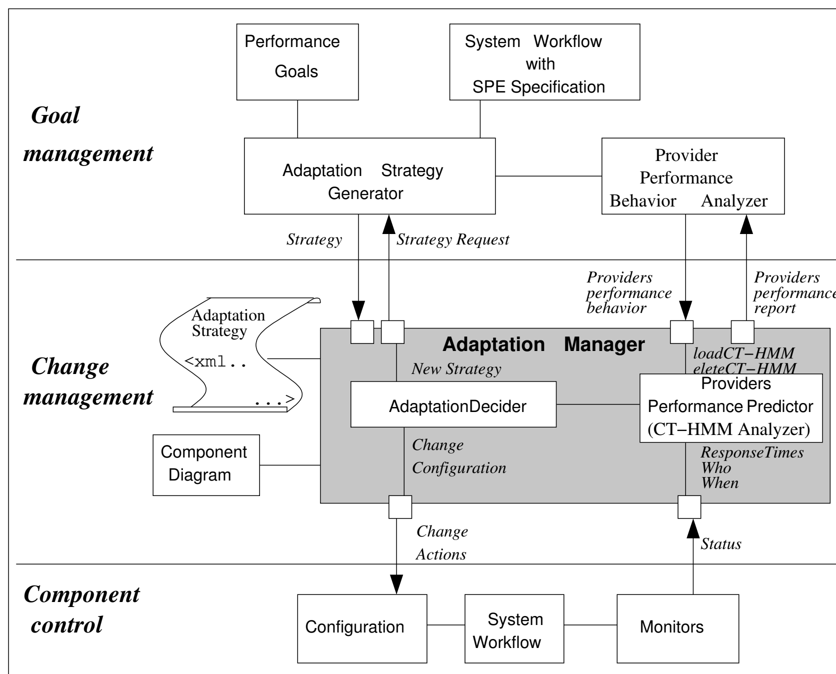


Figure 4.6: Emphasized Change Management layer on the 3-layer architecture adapted to open-world

interfaces when:

- the upper layer *provider performance behavior analyzer* produces a provider performance behavior that was not included in the *Providers Performance Predictor* (*loadCT-HMM*).
- a service provider vanishes off the world (*deleteCT-HMM*).
- knowledge about a service provider behavior is out of date and the *provider performance behavior analyzer* offers an updated model (*deleteCT-HMM* followed by *loadCT-HMM*).

4.5 Conclusion

In this chapter, we have presented an approach, based on HMMs, to predict the performance of SOA providers in the open-world. We have used such prediction to decide the appropriate moments to change the system configuration. We have also fitted the approach in the 3-layer architecture, concretely in its second layer.

The parameterization of the CT-HMMs has not been addressed in this work. However, it is possible to integrate an already existing solution to find the most probable parameters.

4.6 Related work

The motivation of our work in this chapter is shared with the work in [GMMT10], where authors evaluate providers selection strategies. In [GMMT10], comparison between strategies is based on the mean response time the clients achieve. Although our work shares motivation regarding to reach best workflow performance, we take different assumptions. Firstly, we do not rely on user agreements or collaborations to reach a global knowledge about providers performance behavior and their changes, but we assume independent adaptive clients that only concern about their best performance in a selfish way. Secondly, we assume that our requests do not affect the workload of the providers and hence neither their performance.

Regarding the usage of HMM in self-adaptive software systems, work in [WY07] uses them in the provider side to predict requests based on the monitored history of the clients behavior.

The layered architecture for model-driven adaptation explained in [TGEM10] has some aspects in common to ours -beyond the 3-layer architectural view-, such as the dynamic generation of adaptation plans. A key difference is that in our proposal, actions to follow the plans are based on probabilities, since they necessarily come from predictions about properties of third-party service providers that operate in the open-world, and these predicitions are necessarily subject to uncertainties.

Chapter 5

Self-adaptation for Energy Conservation

Energy use is becoming a key design consideration in computing infrastructures and services. In this chapter, we focus on service-based applications and we instantiate the 3-layer adaptation framework to architect a system that is able reduce its power consumption according to the observed workload. We concentrate on the work of the uppermost layer and we propose the generation of adaptation plans that guarantee a trade-off between energy consumption and system performance. The approach to reduce power usage is based on the principle of proportional energy consumption obtained by scaling down energy for unused resources, considering both the number of servers switched on and their operating frequencies. The formal method that helps us for the modeling of the framework concerns is the GSPNs. After presenting the approach, it is applied to a simple case study to show its usefulness and practical applicability.

5.1 Motivation

The constant growth of energy usage in industrialized countries is creating problems to the sustainability of the Earth development. The problem of energy use concerns many fields in human activities: for this reason some new disciplines such as green computing are growing up to study how to consume less energy by providing the same quality of service [Ran10].

As shown in [DKL+08, Ran10], the interest towards efficient use of technology is motivated by some alarming trends showing, for example, that computing equipment in the U.S. alone is estimated to consume more than 20 million giga-joules of energy per year, the equivalent of four- million tons of carbon-dioxide emissions into the atmosphere [Ran10]. IT analysis firm IDC (http://www.idc.com/) estimates the total worldwide spending on power management for enterprises was likely a staggering 40 billion dollars in 2009.

Large computing infrastructures, like data centers, web services hosting or email, in the U.S. consumed the 1.5% of all electrical power in 2006 and it grows at an annual rate of

12% [CJH+11]. Nevertheless, it is possible to observe that they are so complex that some parts become inactive even during active periods. In this chapter, we focus our research on the consumption of the computing infrastructure of the providers of service-based applications. Often when deciding the amount of resources -hardware and software- to include in the platform, worst-case scenarios are considered, which leads to over-provisioning for other scenarios of the system. The result is a static system deployment that wastes part of the available processing infrastructure and consequently causes energy waste.

Therefore, a first direction that can be followed for energy savings is the definition of adaptation plans that can be used to reduce power in time (turn off during idle times) and space (turn off inactive elements). Hence, infrastructures can be dynamically scaled to conserve power with no impact on performance while they match workload demands.

The definition of this adaptation plan is not easy, because the workload is typically variable and unpredictable and because there are also other, possibly contrasting, goals that should be satisfied. Indeed, the ultimate goal of a service provider is to maximize profits from its offered services, while for a client the main objective is to obtain a service with required QoS at the minimum cost. Therefore a suitable adaptation plan should be able to define the best trade-off between energy consumption and QoS offered. The problem is quite complex and -as mentioned in the related work in Section 5.9- there exist in the literature several attempts to propose methods for managing power and guaranteeing the agreed quality of service.

Among the multiple QoS attributes of software to face with energy, in this chapter we choose the system performance in terms of mean response time. The problem of maximizing providers revenues, although important, is not directly tackled. The reason is that we follow the same vision as the one in [CDQ+05], which defends that quality requirements always must be met once contracted. However, other approaches consider that it is fair to violate QoS contracts deliberately in some cases; for example, in case that the penalty paid by the provider to the customer due to contract violation is lower than the investment necessary for meeting the agreed QoS. These methods allow providers to increase their profit at the expense of their reputations.- Nevertheless, the problem is indirectly addressed in this work, since having a strategy that scales the amount of servers, while satisfying the performance requirements, reduces the expenses in the equation $profit = revenues - expenses$.

Proposed Solution In order to reduce energy waste, the processing infrastructure of a service provider can be dynamically accommodated to the actual processing requirements for each scenario. Since the received workload varies frequently and in some cases unpredictably, human intervention to modify the amount of dedicated processing resources is not feasible. So the goal is to have the system aware of its processing resource needs, and able to self-adapt its processing infrastructure to fulfill such needs. Therefore, the objective is to build systems that can autonomously manage their processing resources in order to consume only the power necessary to satisfy their -possibly evolving- performance requirements. These new techniques actually complement the traditional and well-known off-line capacity planning [MA01]. To achieve the objective, in this chapter we instantiate the reference architecture in Chapter 2 to architect a system that is aware of its processing demands, performance requirements and available computing resources. After, we work on the uppermost layer,

i.e., *Goal Management*, and it is proposed a method to generate adaptation plans. These plans tackle the adaptation decisions that decrease as much as possible the system's energy consumption while maintaining the expected performance. The approach also allows the plan regeneration when its execution context changes, which would make the current plan not suitable. The adaptation plan indeed depends on the dynamic variable workload, on the available processing resources, on the application processing demands and on the agreed QoS in terms of performance requirements. To exemplify the deployment of the approach, it is discussed one deployment of interest for the case of software services.

To study the relations among these properties, we follow model-driven techniques to transform design models into analyzable models. In this case, the analyzable models are the Stochastic Petri Nets (SPNs) subnets. Subnets allow modeling the variable workload, the workflow, the processing resources and the logic to adapt the system energy consumption. The considered variables are not new, several works (e.g., [EKR03, CDQ+05]) and a survey [BR04] exist on this topic.

As recognized in [CJH+11], queuing models, category of which SPNs are an example, are ideal to predict runtime trade-offs between performance and energy use. Moreover, queuing models have been largely validated during the last decades and we can be absolutely confident in the results they produce, which may free the modeler from the need of validating the model as long as it accurately represents the target system. This is an advantage regarding ad-hoc models, heuristics or equations when used to model complex behaviors, since they really need extensive validation to prove that the predictions they obtain actually match the real measurement. In contrast, queuing models have been accused of being difficult to construct. In this regard, we try to keep our models as simple, repeatable and scalable as possible and we propose tools to automatically construct them.

To generate a Petri net that represents the whole system behavior, we put together the previously mentioned subnets. Hence, this analyzable SPN includes fine-grained information regarding: mean execution times of internal activities; resource usage of activities; resource competition for passive resources (e.g., buffers) which generates "waits" and makes the system performance not scaling linearly with frequency; and resource competition for active (processors) which are the basis for power consumption.

The SPN evaluation, carried out with the GreatSPN tool [Gre], gives results about the suitability of the adaptation plan (in terms of whether it deteriorates performance results) and how much energy it saves. Moreover, we define a parametric Petri net that can be evaluated to discover which are the best parameters to tune the adaptation plan, in order to save as much energy as possible.

Motivating Example We describe a kind of system for which can be applied our approach. Consider a company that develops software services which are offered in the Internet, some of them for free while others can get subscription rates. Irrespective of the implementation, the services follow a Service Oriented Architecture (SOA). The company maintains a homogeneous computing infrastructure, around hundreds of servers, which deploys the services. These services are used all around the world and they can receive thousands of requests per minute at certain times of day, however it is also possible that the workload decreases at certain hours considerably. When the workload is in a peak the infrastructure has to be fully

operative and each service will be replicated in as many servers as necessary to support the quality of service the company promises. On the contrary, when the workload is low, most of the servers can not be necessary at all. Therefore, the company needs an integral *software* solution, beyond the traditional load balancer, that switches on and off the servers to adapt the infrastructure to the workload dynamically. We argue that if the software solution follows the architecture described in this chapter, the infrastructure can achieve the advantages previously discussed, i.e., a good trade-off between QoS and energy conservation.

The remainder of the chapter is organized as follows. In Section 5.2 we present the instance of the 3-layer architecture for self-managed systems for the management of energy and performance. The proposed SPN models for dynamic variable workload and energy consumption are presented in Sections 5.3 and 5.4, respectively. The trade-off between energy consumption and the fulfillment of the performance goal is presented in Section 5.5. Section 5.6 discusses a suitable deployment of the architecture and presents evaluation through an example, which is developed step by step to help practitioners to learn the proposal. The evaluation continues in Section 5.7 to experiment with variable workload. Section 5.8 draws some conclusions and provides pointers to on-going work. Related works presented in Section 5.9 complete the chapter.

5.2 Architecture

In this section we instantiate the reference architecture for self-managed systems presented in Chapter 2 for software systems whose adaptation goal is to save energy. Figure 5.1 describes our proposal identifying responsibilities for each layer and the necessary software modules that can carry out them.

The *Component Control* layer accomplishes the application function of the system, in our case the workflows of the software services the infrastructure deploys. The *software services* modules represent the executable files of these software services. They are the instance of *software application and infrastructure* in Figure 2.1. Note that they are replicated, really to represent several services but also several running instances of each service. Each running instance, which manages requests until its maximum capacity, will execute in a server of the infrastructure. Each server can host several running instances. The *Component Control* layer also features a *HardwareController* and a *LoadMonitor* software modules. They are the instances of *sensors* and *actuators* in the general architecture and they include facilities to report the current status of the processing infrastructure and to support modifications on it. The *HardwareController* implements: a sensor functionality for communicating with its upper layer to inform the current state of the servers (e.g., booting completion), and an actuator functionality to receive orders to reconfigure the server infrastructure (increase/decrease frequency or switch on/off of servers). We think of it as a software module that manages the servers through the Wake on LAN (WoL) facility. The *LoadMonitor* monitors current system workload and informs to its upper layer when the workload exceeds some thresholds, i.e., a problem to solve. Thresholds of interest were previously identified by the upper layer to this module.

The *Change Management* layer executes actions to handle the new situations reported by the lowest layer. It is made by a software module, the *EnergyManager*, and its input

file, called the *Energy-aware Adaptation Plan*. The *EnergyManager*, which is the instance of the general *adaptation manager*, is informed of the system workload and the status of the processing infrastructure and it uses the energy-aware adaptation plan to decide *when* to reconfigure the infrastructure and *how* to carry it out. It orders reconfigurations when it recognizes a non optimal one: either the system load is low and the performance goal could be fulfilled using less resources, or the load is high, requiring more capacity to satisfy the goal. The energy-aware adaptation plan is received from the uppermost layer, either on demand or when the uppermost layer decides to change it (e.g., because system goal changed).

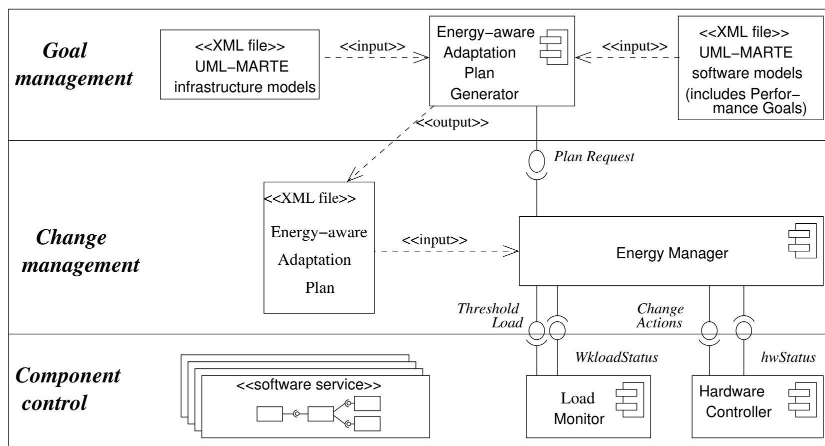


Figure 5.1: KM-3L adapted to energy management

The uppermost layer, *Goal management*, consists of time consuming computations to produce a plan to achieve the goal of the framework, in this case the goal is to allow the infrastructure to satisfy its performance goals while it spends as less energy as possible. This layer is made of a software component, called *Energy-aware Adaptation Plan Generator*, and a set of software and systems models, think of them as UML models, so we can assume we have their XML representation. These files are inputs for the *Energy-aware Adaptation Plan Generator*. The *Energy-aware Adaptation Plan Generator* creates plans following a model-driven approach where the software and system models are transformed, following the proposal in [LGMC04], into an analyzable Stochastic Petri Net (SPN) model.

The software models will represent the workflow logic of the services deployed (e.g., UML activity diagrams). These models also contain the performance characteristics of the software service (e.g., using the MARTE [Obj05] profile to annotate the previous diagrams). The performance characteristics include: expected workload, which is the changing environment in this work; performance goals of the service, which are the instance of the general *extra functional goal*; processing demand and execution probabilities of the activities, and resource sharing, which are a part of the general *system models*. The second part of *system models* is stored in the infrastructure models, which represent the processing platform

(e.g., UML deployment diagrams). These infrastructure models include: number of available servers, its processing capabilities w.r.t. power consumption and mechanisms to change power consumption.

The *Energy-aware Adaptation Plan Generator* carries out the SPN analysis to produce the plan, then obtaining the maximum load the configuration can manage while the required performance goal is accomplished. Section 5.5 describes how to generate a plan. Once the plan is generated the task of the *Energy-aware Adaptation Plan Generator* is not finished yet. It performs a plan evaluation to predict system behavioral characteristics using the generated plan. To execute such prediction a new SPN will be created starting from the previous one and adding information regarding: variable workload, platform energy consumption and the adaptation plan itself. New SPN sub-models will represent each one of the previous concerns, which are explained in detail in subsequent sections (Sections 5.3, 5.4 and 5.5 respectively).

Once the system behavioral prediction has been derived, the goal of this layer is to periodically check whether the plan is suitable, which means verifying:

- whether the system is behaving as expected.
- whether the models of the workflow, workload and the platform are close to the real behavior. For example, the assumed values for the workflow activities could change due to software upgrades.

If some of these issues are not adequate, this layer will update model parameters and will regenerate the plan.

5.3 Workload modeling

In order to carry out a proper model-based analysis of system's behavioral properties, we first need a model-based representation of the workload the system is managing. This is a not trivial concern since dynamic systems should be able to cope with highly variable workloads with temporal dependencies.

Classical techniques to model workload, such as those based on phase-type ([Buc03], [PT07]) or exponentially distributed inter-arrival time of requests, do not consider dependencies and correlations between inter-arrival times. However, these concepts are crucial if the system to be evaluated takes into account its incoming workload to decide its operational mode. Therefore, to evaluate this type of self-adaptive systems, our workload models have to be able to represent both the variability and the temporal dependency. To this end we adopt a SPN model since SPN have been largely used in the literature for this purpose. Indeed, since our analyzable model of the system is based on SPNs, it is an advantage to have the workload model represented in the same formalism in order to be integrated with the rest of the system model.

To study the workload characteristics, we first consider different granularities regarding workloads' time scale: In the long-term (e.g., time span of a week), it is possible to devise a pattern (or a distribution) that fits the variable workload. But in the short-term (e.g. time span of several seconds), the high variability of the workload makes prediction very challenging. In our architecture, a solution could pass through waiting until the *Load monitor* module has

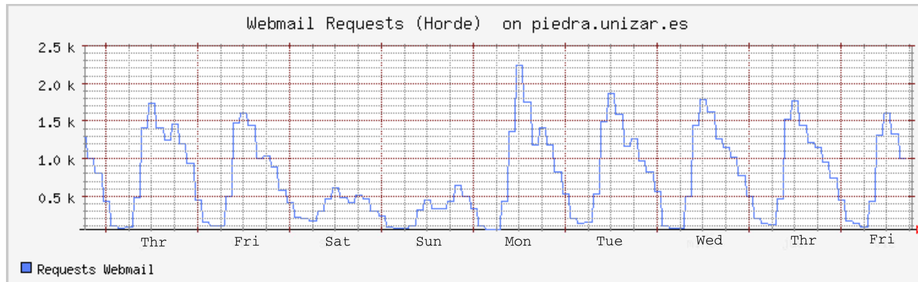


Figure 5.2: Webmail server, weekly supported workload

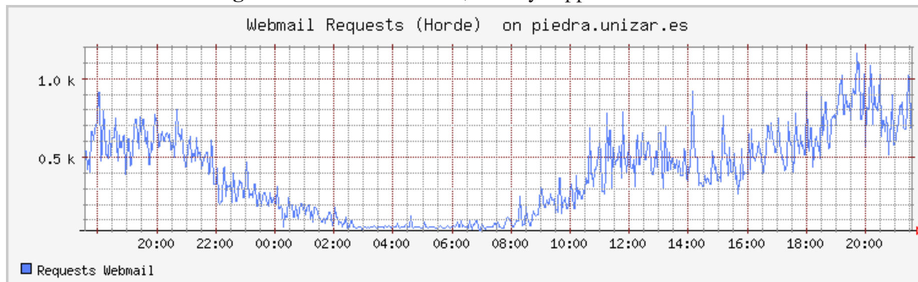


Figure 5.3: Webmail server, daily supported workload

monitored enough data to acquire a long-term view and then proceed to adapt. However, such behavior will delay adaptation decision far away from the point in time it has been needed. Therefore, it is necessary a prediction method that only uses short-term monitored data can quickly infer the current workload in the long-term (and then also infer the near future expected workload). Obtaining such prediction method is a real challenge since it should manage multiple long-term variables and obtain their current values managing only partial, short-term, information.

When studying the workload, we observe that long-term arrival rates of requests for service can be clearly separated in several states. Following this assumption, our workload model will contain several states representing each one a concrete arrival rate. Figure 5.2[1] represents a real variable workload supported by a mail server (which receives around one million requests per day and 30,000 login operations). In the Figure, we can appreciate several states: (i) *night* with an arrival rate close to zero and duration around 8; (ii) *working-hours* with an arrival rate around 1,500 requests/minute and duration of 16 hours; (iii) *peak* which is sporadic, short (it lasts for around one hour), it takes place only during working hours and can reach an arrival rate around 1,800 requests/minute; and (iv) *weekend* showing an arrival rate around 500 requests/minute and lasting roughly 16 hours. However, the short-term arrival rate of requests is not so regular, making the prediction of the workload state a challenging

[1]Figure taken from https://piedra.unizar.es:8080/public/monitor during the week of 4-12 November 2010.

task. To illustrate such challenge, Figure 5.3 shows the variability of the workload from Sat 20:00 to Sun 20:00.

To model such workload, we define the $SPN_{workload}$ with a shape like the one in Figure 5.4. The theory to automatically estimate the parameters of the underlying Markovian model can be found in [CMCS12]. For the presented example of the webmail server, we set parameters manually to pay attention on the resulting model rather than in the modeling process. The SPN models both the long-term and the short-term workload behavior and it includes:

- as many places as workload states. A token in a place means that the system is receiving requests with the arrival rate associated with that state. Therefore only one of these state places can be marked.

- a timed transition for each state. Such transitions are bidirectionally connected to state places. These transitions inject the workload to the beginning of the workflow. Their firing rate corresponds to the expected workload in each state. In Figure 5.4, firing rates are denoted as λ_{state}. Since these transitions are linked to state-places, only one of them can be enabled.

- a set of timed transitions to model the state mean sojourn time[2] and probabilities of change between states. For example, transitions T_{N-Wo} and T_{N-We} model the mean sojourn time in *night* state, the sum of their rates must be $8hours^{-1}$. Moreover, to model the change state probabilities, i.e., five changes per week from *night* to *working* and two changes per week from *night* to *weekend*, it is required that $\lambda_{T_{N-Wo}} \cdot 8hours = \frac{5}{7}$ and $\lambda_{T_{N-We}} \cdot 8hours = \frac{2}{7}$, which lead to $\lambda_{T_{N-Wo}} = 5.715hours^{-1}$ and $\lambda_{T_{N-We}} = 2.285hours^{-1}$. [3]

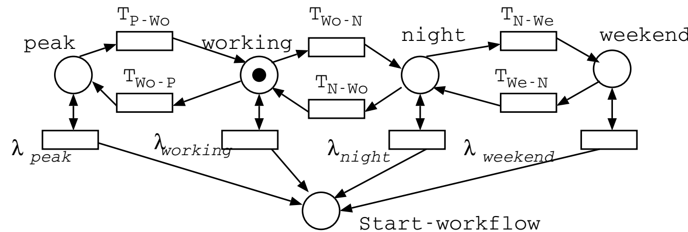


Figure 5.4: $SPN_{workload}$ model

The derivation of the unknown parameters λ of the $SPN_{workload}$ is based on the Markov Arrival Processes (MAP) theory, and in particular on a type of MAP, those called Markov Modulated Poisson Process (MMPP). There exist theories to automatically create and parameterize these models. Since these theories are out of the scope of this work, we have presented

[2]By *mean sojourn time* we mean the average time the system spends in a given state.

[3]It has been only considered the mean amount of changes between states, so, it has not been considered that the two changes from night to weekend per week should be consecutive.

the $SPN_{workload}$ parameterization as a manual process. A detailed description of MAPs and MMPPs can be found in Chapter 6, together with a parameterization technique for a special kind of workload.

We have simulated the behavior of the Petri net and compared the obtained results with the real workload in Figures 5.2 and 5.3. Figure 5.5 illustrates the simulation results (token arrivals to place `Start-workflow` w.r.t time): the shape and pattern match with the long-term view of the real server in Figure 5.2. The long-term view in Figure 5.5 has been achieved by counting the events generated in slots of 20 seconds. Moreover, zooming in the simulation results (right part of the figure) we can also observe the high variability in the workload that the system is receiving, which makes the workload state prediction be a challenge. See for example that the value marked with circle is higher than the value marked with the triangle, while the long-term view workload supported by the state of the circle (*working*) is lower than the long-term view workload supported in the state of triangle (*peak*). The short-term view in the figure has been achieved by counting the events generated in slots of 1 second.

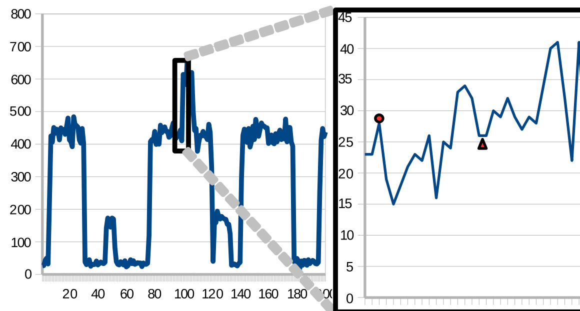


Figure 5.5: Simulation of the $SPN_{workload}$ model

It is worth noting that the $SPN_{workload}$ part will be "isolated" from the rest of the SPN model that represents the system. The unique element in common between them is *start-workflow* place. In this place $SPN_{workload}$ holds tokens that represent execution requests from users. In turn, the SPN that models the system will delete such tokens and will start a system execution for each of them. Therefore, the rest of the SPN cannot get information regarding which is the active state in each moment (i.e., in which place $state_i$ the token is) or regarding the firing of any transition $T_i \in SPN_{workload}$. At most, the rest of the system can monitor the token generations in *start-workflow* place during a certain period to try to predict the expected workload.

5.4 Energy modeling and analysis

In this section it is proposed a SPN model that allows the evaluation of the variables related to *energy consumption* and *frequency of the servers*, both taken into account in the adaptation plan. This SPN, in Figure 5.6, also models the transient state of servers, from switch off to

on and vice-versa, which means to embed actions defined in the adaptation plan to manage power consumption. Some places in the SPN will be shared with other subnets (e.g., the workload subnets in previous Section) to make the final SPN model. Indeed, the evaluation of the variables herein presented will be carried out in this final SPN.

From Figure 5.6 we see that when a switched off server receives the `SwitchOnEvent` it begins its `Booting`. That booting process lasts for $T_{startup}$ time units. When the booting is finished, the server is operative to receive requests and the completion is notified to the energy manager through a token in `BootedEvent` place. When an operative server receives a `SwitchOffEvent` it starts its shutting down process. First of all, its representative token in `OperativeServers` is deleted, meaning that it is no longer available to receive new service requests. A server changes its state from operative to a state `WaitForRequestsCompletion`, where it is finishing its ongoing requests. When all ongoing requests are finally served, it starts the `Halting` process which lasts for $T_{shutdown}$ time units. After that, it joins the pool of `SwitchedOffServers`. Tokens in `SwitchOnEvent` and `SwitchOffEvent` come from the energy manager. Tokens in `OngoingRequests` are generated by the workload balancer and deleted when a request finishes its execution, these tokens store information about the server that is executing the request. Tokens in `OperativeServers` are looked up by the workload balancer when it has to decide the target server for a request.

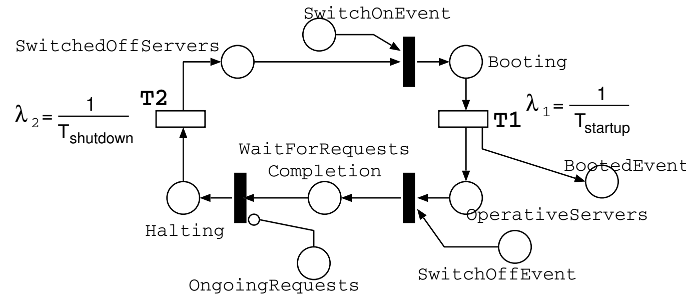


Figure 5.6: SPN modeling the states of servers

Regarding the *energy consumption variables*, we will evaluate in the final SPN the following:

1. Mean power consumed by switch on and off processes,

$$W_{on-off} = C_{startup} \cdot \chi(T1) + C_{shutdown} \cdot \chi(T2)$$

where $\chi(Ty)$ is the mean throughput of transition T_y. $C_{startup}$ and $C_{shutdown}$ respectively represent the energy consumed by the server at start-up and shutdown.

2. Minimum power consumption of a server, $W_{standby}$, includes all constant consumptions that do not depend on the working frequency. Mean aggregated power consumption of

servers is

$$W_{AggreagedStandby} = W_{standby} \cdot (E[\#OperativeServers]+ \\ E[\#WaitForRequestsCompletion])$$

where $E[\#Px]$ is the mean number of tokens in place P_x.

3. Maximum power consumption of a server, W_{max}, considers when a server is busy and working at its maximum frequency.

4. Since voltage supply limits the maximum operative frequency of the circuit approximately to a linear factor, then following [EKR03, CDQ+05], we merge dynamic frequency scaling and dynamic voltage scaling, and we obtain that power consumption is proportional to the cube of the working frequency. Therefore, power consumption of a server in an operational frequency will be

$$W_{freq_i} = (W_{max} - W_{standby}) \cdot (opFreq_i)^{3}.$$

Finally, the total amount of power consumed by a server working at frequency $OpFreq_i$ is

$$W_{server_i} = W_{standby} + W_{freq_i}.$$

In the SPN, the mean power consumption of a single server is calculated as

$$W_{mean} = \sum_i W_{server_i} \cdot P(\#Frequency = i+1),$$

where $P(\#p = n)$ means that the probability of the number of tokens in place p is equal to n. Meaning of place $Frequency$ is unveiled in the following.

Regarding *servers processing frequency*, dynamic frequency and voltage scaling allow varying working processors performance and to reduce their power consumption. Although the working frequency could ideally range between 0 and 100% of processor capabilities, real working frequencies are usually discretized. Therefore, as in [CDQ+05], we assume that the actual server frequency is restricted to a value within a set of operational frequencies $FreqSet$. We consider $FreqSet$ made of a base frequency $BaseFreq$ and increments $BaseInc$. Therefore

$$FreqSet = \{OpFreq_i\} \mid OpFreq_i = BaseFreq + i \cdot FreqIncr \\ \wedge (i \geq 0) \wedge (OpFreq_i \leq 100\%).$$

For example: $BaseFreq = 50\%$, $BaseInc = 10\%$ and $FreqSet = \{50\%, 60\%, 70\%, 80\%, 90\%, 100\%\}$. Advancing a description of the Petri net model in Section 5.5 (Fig. 5.9), tokens in place `Frequency` will represent servers processing frequency. It will contain from 1 to $|FreqSet|$ tokens (from minimum to maximum frequency). A reconfiguration in the server frequency will obviously change the number of tokens in this place.

To keep the model simple, we do not model other variables related to power-aware adaptation such as savings in the cooling system.

5.5 Performance and energy trade-off

This section explains the process of creating an energy-aware plan that cares of performance requirements. To ease the explanation, we divide the process into two steps: the former to generate a basic plan, Subsection 5.5.1, while the latter to optimize it, Subsection 5.5.2. Subsection 5.5.3 proposes a Petri net model for the plan described in Subsection 5.5.2. Finally, Subsection 5.5.4 presents a Petri net that results from merging all the PNs obtained so far. That model will be useful to carry out a trade-off evaluation (performance and energy) of the system.

5.5.1 Generation of basic-plan

An energy-aware adaptation plan will be a set of system *configurations*, that meet the performance goals using minimum energy, and *actions* to change among configurations. A configuration defines the number of active servers as well as the frequency they are working at. Hence, actions to change a configuration will just mean to switch on/off servers and/or change their working frequency. Using the number of servers and their working frequency, the power in each configuration can be calculated.

A configuration also identifies a threshold that corresponds to the maximum system load the configuration can manage. Energy manager uses information in the plan to accommodate the system configuration to the most suitable regarding the current number of requests (system load). System load is an information the plan receives from the lower layer, which indeed monitors the system.

In the following the process to generate the energy-saver adaptation plan is explained (Table 5.1 will help the process understanding).

1. Generate a SPN model of the system workflow which includes the required processing demands. Set the capacity of servers to a minimum (e.g., in Table 5.1, one server, k=1, at its minimum frequency, $OpFreq_0 = 50\%$).

2. Evaluate the SPN to discover the mentioned threshold, i.e., the maximum load ($Nrequest$) it can manage while the performance goals are satisfied. Compute power consumption (W_{server_i}) for this configuration. (In Table 5.1, 27 and 16.3 respectively for the first case.)

3. Increase the server frequency (which means to modify the SPN) and go to step 2. Repeat this step for all the frequencies the system has to manage, i.e., $|FreqSet|$.

At this point we have completed one row of the table. It is natural to assume that, if a server working at frequency $OpFreq_i$ can manage $Nrequests_i$ and spends W_{server_i} power, k independent and concurrent servers working at the same $OpFreq_i$ are able to manage $k \cdot Nrequests_i$ and they spend $k \cdot W_{server_i}$ power. Applying this, we compute the rest of rows in the table multiplying the first row values by the number of servers that represent each row. We will consider as many servers as available in the infrastructure. As a result, we have generated a table that contains all possible system configurations and, for each configuration, its power consumption and the load it is able to manage (the complete table is not displayed).

		Percentage of frequency, $OpFreq_i$					
		50%	60%	70%	80%	90%	100%
k=1	*Nrequests*	27	32	37	43	48	54
	W_{server}	16.3	18.4	21.3	25.1	30	36.2
k=2	*Nrequests*	54	64	74	86	96	108
	W_{server}	32.6	36.8	42.6	50.2	60	72.4

k means number of active servers

Table 5.1: Information required to create an adaptation plan

Using data in the generated table, Algorithm 5.1 can be applied to generate the basic adaptation plan. This plan contains an ordered list of a subset of possible configurations (called *suitable* configurations) as well as threshold values indicating the moment to change from one configuration to another.

As a result we can distinguish two kinds of adaptations: those that only require to change the frequency and those that require to change the number of working servers -most probably, together with their frequency.

An example of configuration in Table 5.1 is the system working with only one server ($k = 1$), with frequency 60% and then with thresholds 27 and 32 requests, in this case the power consumption is 18.4. System load ranging between 0 and 48 can be managed by only one server and changing only the frequency. However, when the number of requests exceeds 48 it will be better to change to a configuration with 2 servers and frequency at 50% since power consumption is 32.6 instead of 36.2 offered by the configuration that only changes frequency. So, the configuration that uses one server at 100% will never be used. Figure 5.7 shows a basic adaptation plan in a chart. It depicts reconfiguration points in function of the workload, considering a six-server infrastructure.

5.5.2 Reconfiguration rate mitigation

The basic-plan suffers periods with high rates of switching on and off of the servers, which is a real drawback for two reasons. First, the time spent in booting and halting can be too high w.r.t. the real working time, then the energy spent in switching tasks is not spent in serving requests. Second, the more the switching rate, the more the wear and tear of servers.

To reduce the number of switch on and off of the servers we propose to use reconfiguration limits with hysteresis. In other words, the *Nrequests* threshold value indicating when the system changes between two neighboring configurations will not be unique but composed of a couple of numbers, $Nrequests^{dec}$ and $Nrequests^{inc}$, according to whether the system tendency is reducing its power (moving from the high energy consuming configuration to the lower) or increasing it (moving from the lower consuming configuration to the higher). Therefore, the association between the supported load and the system configuration will not be unique.

Algorithm 5.1 Basic-plan generation

Require: Table with $Nrequests$ and W_{server} (dimension KxF)
Ensure: Basic adaptation plan.
1: **set** $k = 1$ {considered number of servers, row index}
2: **set** $f = 1$ {considered frequency, col index}
3: **set** $Plan \leftarrow EmptyPlan()$ {create empty plan}
4: **set** $currentConf \leftarrow Table[k][f]$
5: $Plan \leftarrow AddToPlan(Plan, CurrentConf)$
6: **set** $cadidateFreq$
{Search rest of suitable configurations until finish the table}
7: **while** $k < K$ **do**
8: $cadidateFreq \leftarrow GetBestInRow(k + 1, currentConf)$
9: **if** $IsBetterToContinueWithSame$ $Servers(k, f, candidateFreq)$ **then**
10: $f \leftarrow f + 1$ {Next configuration increases frequency}
11: $currentConf \leftarrow Table[k][f]$
12: **else**
13: $k \leftarrow k + 1$ {Next configuration increases servers}
14: $f \leftarrow candidateFreq$
15: $currentConf \leftarrow Table[k][f]$
16: **end if**
17: $plan \leftarrow AddToPlan(Plan, currentConf)$
18: **end while**
{Add last row of table to plan}
19: **while** $f < F$ **do**
20: $f \leftarrow f + 1$
21: $plan \leftarrow AddToPlan(Plan, Table[K][f])$
22: **end while**
23: **return** Plan

The meaning of these new limits are: $Nrequests_s^{inc}$ corresponds to the threshold amount of requests to change from configuration s to $s+1$. $Nrequests_s^{dec}$ corresponds to the threshold amount of requests to change from configuration $s+1$ to s. In Figure 5.8, bold continuous line shows the $Nrequests^{inc}$ values, which are very similar to the previous $Nrequests$ while bold dashed line depicts $Nrequests^{dec}$. In that graph, the hysteresis length is equal to 2 steps, i.e, the dashed line is moved two configurations above the continuous line. Therefore, $\forall s \in \{2..S\}$ $Nrequests_s^{dec} = Nrequests_{s-2}$. For example, supposing that configuration s is the one that uses three servers working at 60% of its frequency, and looking at the change from configuration s to $s+1$ (i.e., use three servers working at 70%), $Nrequests_s^{inc} = 94$. However, looking at the change from configuration $s+1$ to s, $Nrequests_s^{dec} = 74$; value which corresponds with the previous $Nrequests_{s-2}$.

Therefore, the higher the hysteresis length, the lower the reconfiguration rate and the less the wear and tear, but higher the mean power consumption, since the system spends more

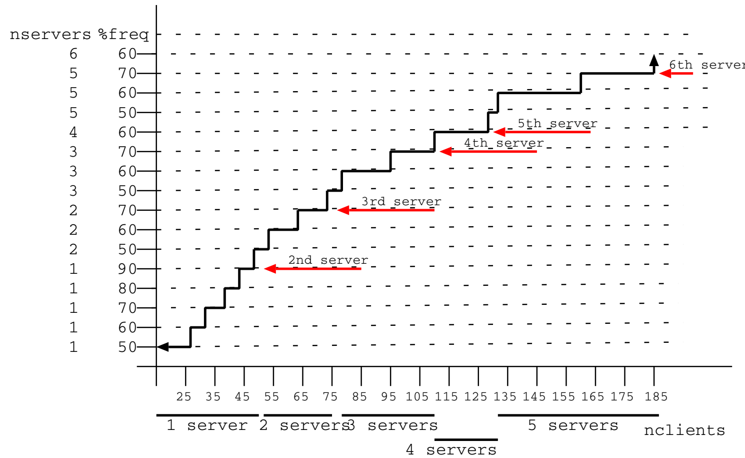


Figure 5.7: Graph for system reconfigurations

time in a configuration that consumes more energy than necessary to deal with the received workload. There is an example of the trade-off between these characteristics in Section 5.6.

It is possible to observe small differences between bold continuous line in Figure 5.8 and black line in Figure 5.7: these are due to corrections made when the reconfiguration involves to turn on a new server. These corrections are intended to mitigate the non quality satisfaction during the booting time of the newly switched on server. We start to switch on a server few moments before it will compulsorily need to maintain the required quality. This helps to have it already booted and completely operative when it has to be used. Among the multiple manners to decide how much the booting moment should be brought forward, we choose to calculate it as a proportion of the step length called Bring Forward Proportion (BFP). Thus, when the system is in a configuration $Conf_j$ such that the immediately consecutive $Conf_{j+1}$ uses one server more, the adaptation order will take place when system load reaches $Nrequests^{inc}_{Conf_j} = Nrequests^{inc}_{Conf_{j-1}} + \lfloor (Nrequests_{Conf_j} - Nrequests^{inc}_{Conf_{j-1}}) \cdot BFP \rfloor$.

As an example, let us consider the difference between the bold continuous line in Figure 5.8 (in $Nrequests = 111$) and black line in Figure 5.7 ($Nrequests^{inc} = 96 + \lfloor (111 - 96) \cdot 0.75 \rfloor = 107$) with a BFP value equal to 0.75 and focusing on the moment to order the switching on of the 4th server.

Thus, without hysteresis and BFP, three servers were used when the load of the system ranged from 74 to 111. With the new improvement, three servers can be used to manage from 52 to 107 requests, but what happens concretely, is that two or three servers are used to manage from 52 to 74 requests, exactly 3 servers for the range 74-76, three or four servers to deal with requests from 78 to 96 and three, four or five servers manage requests from 96 to 107.

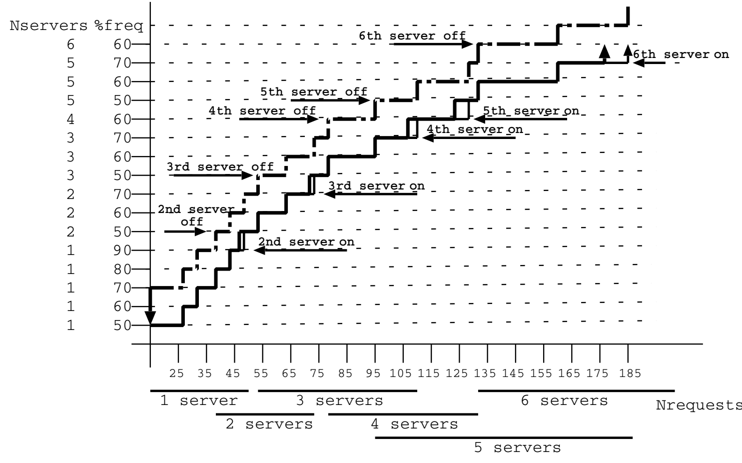


Figure 5.8: Graph for reconfiguration (with hysteresis)

The decision to set a suitable value for the hysteresis proportion is studied in Section 9.4 by means of the evaluation of an example with different proportion values.

5.5.3 Petri net model of a plan

The Petri net in Figure 5.9 models the system reconfigurations that an energy aware adaptation plan could carry out, in this example there are depicted configurations $Conf_0$, $Conf_1$, $Conf_2$, $Conf_s$ and $Conf_{s+1}$ of the plan. Places representing configurations, $Conf_i$, are in mutual exclusion and can contain at most one token.

Transitions in the right hand side (t_4 and t_5) allow to upgrade the power of the system changing to a configuration that increments its `Frequency` when the system is supporting a load that exceeds the configuration threshold $Nrequest^{inc}_{Conf_i}$ (weight of the test arc linked to `systemLoad`). Transitions in the left hand side (t_1 and t_2) allow to downgrade the power of the system changing to a configuration that decrements its `Frequency` when it receives less than $Nrequest^{dec}_{Conf_i}$ requests, in this case an inhibitor arc (those having a circle at the end) prevents the firing of the transition when the number of tokens in `systemLoad` is more than $Nrequest^{dec}_{Conf_i}$. `SystemLoad` place will be filled by the workload subnet (the subnet shown in Section 5.3, Figure 5.4), and its tokens removed by a timed transition with infinite server semantic and firing rate $\frac{1}{MonitoredTimeSpan}$, so it accounts for the number of requests the system has received during the lasts *monitoredTimeSpan* seconds.

Some downgrades in the system configuration imply to increment the frequency and to decrement the number of servers, transition t_3 represents them. In this case, the `Frequency` is increased with the difference of frequency between configurations ($Freq(Conf_i) - Freq(Conf_{i+1})$). While the number of servers is decremented sending an event (token in `SwitchoffEvent`) to start the switch off process.

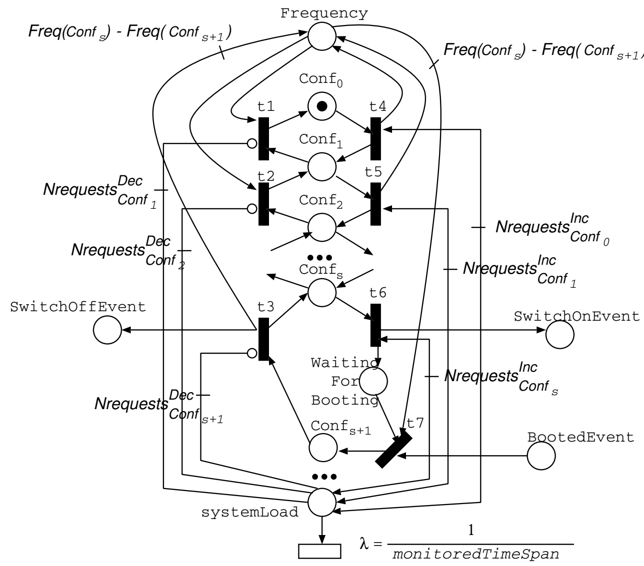


Figure 5.9: Petri net modeling the adaptation plan behavior

On the other hand, some upgrades of system configurations imply to decrease frequency and increase the number of servers, they are trickier and need of two transitions, in the example t_6 and t_7. In this case, the change of frequency and the switch on of the servers cannot be concurrently executed since switch on entails booting time. So if the frequency is changed when the new servers have not been yet added (servers are booting), the servers currently working will be the ones suffering the frequency change and they will provoke a transitory quality degradation of the system instead of its power enhancement. Then we split up the upgrade process in two steps. In the first, t_6 orders the system to switch on the server (tokens in `SwitchOnEvent` and `WaitingForBooting` places). During this booting time the system works at the frequency in the source configuration (no degradation). When the server is already booted (token in `BootedEvent`), the frequency is decreased using transition t_7, and the system reaches the new configuration.

5.5.4 The Petri net for trade-off evaluation

The Petri nets for the adaptation plan, the workload, the state of the servers and the software service are merged to create a new one where to carry out the proposed trade-off analysis. Figure 5.10 depicts an abstract view of this Petri net, where the places that are interfaces clearly emphasize how interact the nets. Although we do not present in this chapter a Petri net of a software service, Figure 5.12 illustrates the workflow of a software service, and we obtain the corresponding Petri net automatically, using ArgoSPE [GMM06].

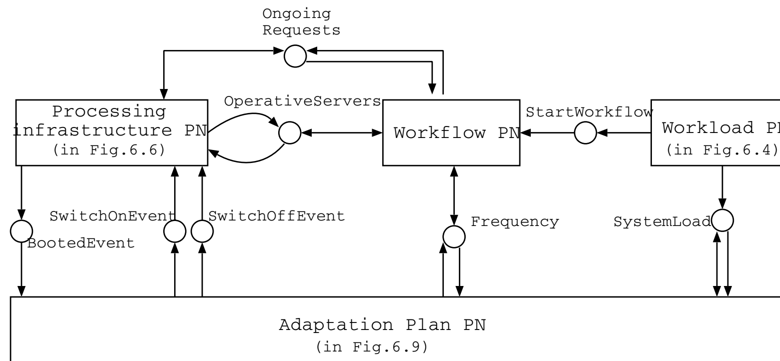


Figure 5.10: Abstract view of the Petri net for evaluation.

5.6 Deployment and evaluation

In this section we present a possible deployment of the modules in the architecture in Figure 5.1. The UML deployment diagram in Figure 5.11 depicts a deployment in which the computing platform is made of `servers`, where the software services in the Component control layer are deployed. The bottom layer of the architecture is completed with the `LoadMonitor` and the `HwController` which are software modules that accommodate in the hardware that receives the requests from clients, also separated hardware could be used. The software module that comprises the Change management layer, `Energy Manager`, is deployed in a separate hardware that only communicates with the other two layers for sending and receiving the orders and information specified in the architecture. Finally the `Adaptation Plan Generator`, which is a software that evaluates SPNs as explained in Section 5.5, is deployed in a high performance computing platform to create the plans on demand, this service could be even provided by a third-party in the cloud.

5.6.1 Evaluation framework

The SPN in Figure 5.10 represents all the elements in the deployment, although some implementation was required to carry out evaluation. The requests of the clients are modeled as proposed in Section 5.3, which confer us the advantages previously presented as well as the choice of performing a plethora of experiments as discussed in Section 5.7. The actions of the `HwController` are embedded in the SPN in Figure 5.6. The software services are simulated by the SPNs that represent them, note that the main interest is to simulate the time they spend, which is accurately represented by the timed transitions of the SPN. These SPNs are obtained from the UML models of the software services using the ArgoSPE tool [GMM06]. Regarding the middle layer, Change management, it is embeded in the SPN in Figure 5.9 which represents the adaptation plan. We have implemented a java program that creates the basic plan, evaluating PNs and applying Algorithm 5.1. The evaluation was carried out using

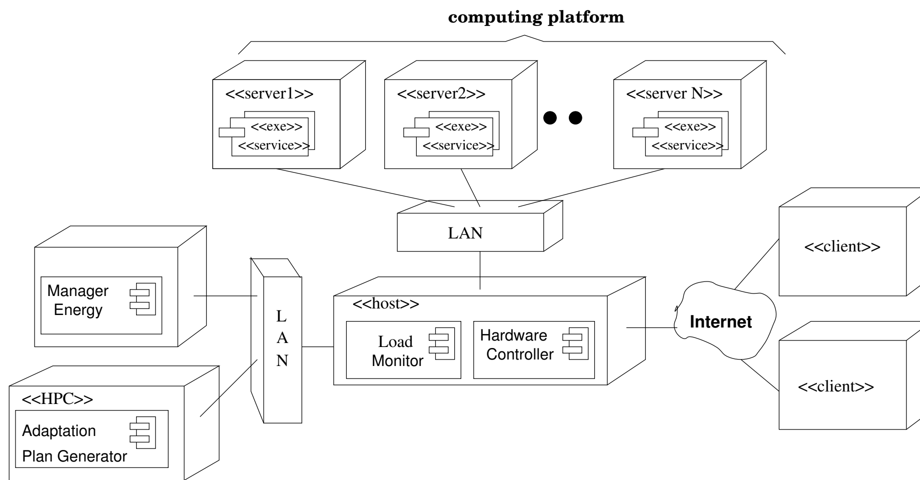


Figure 5.11: Deployment of the reference architecture

the GreatSPN tool [Gre]. The program also creates the plan with hysteresis. The resulting plan gives the parameters for the SPN in Figure 5.9. Hence, we have created a model-based framework, that being able to evaluate our proposal, frees us of developing this expensive deployment, specially in regard to acquire or rent a real computing infrastructure.

Being our purpose a model-based evaluation, we consider interesting to summarize the differences between it and an hypothetical evaluation carried out using the deployment in Figure 5.11:

- We do not have "real" clients but a model of workload. However consider that this part of the deployment does not belong to the architecture, i.e., to our contribution. Moreover, we have shown a method in this chapter to appropriately leverage the workload.

- We have not traded an expensive computing platform made of hundreds of servers. However our SPNs models carefully represent the workload they support and our plan considers their consumption, frequency and booting and shutdown times.

- The `HwController` has not been implemented since we have not the computing platform it manages. However, this task just means to program the WoL facility of the servers and the remote control of the frequency.

- The `Load Monitor` is not necessary in our evaluation since the workload is generated by our model.

- The `Adaptation Plan Generator` has been implemented for our model-based evaluation and it could be reused in the deployment in Figure 5.11.

5.6.2 Example of evaluation: relay mail server

The model-based framework above described has been carried out to evaluate a simplified version of a relay mail server, a kind of system very common for enterprises and institutions. Relay servers use to be replicated to cope with highly dynamic workloads usually being a few the number of replicas, except for extremely large mail providers.

The server receives requests, to route mails to destinations, from both external and local users. First activity is to accept the service. For example, mails from local users are allowed to be delivered to anywhere, while external users could only be allowed to send mails to local users, then avoiding *open-relay* risky configurations. For accepted mails, the relay analyzes the content regarding security, trying to mark viruses, spam or phishing. Safe mails are delivered with a header indicating the analysis result. Mails containing viruses are rejected. The destiny of safe mails can be either an external relay server, the one of the addressee, or the own company mail inbox server. Finally, the relay server writes a log about the operations performed, time stamps and related information (indeed, this kind of logs have been very useful for our research in Section 5.3). Figure 5.12 depicts the workflow, using UML, as well as the performance information, in this case annotated with the standard MARTE [Obj05] profile: a) mean host demand for each operation and b) system routing rates as probabilities. Host demands annotations assume the server working at its maximum frequency. The performance requirement states that mean response time for a legitimate request should be less than two seconds.

Workload model For the sake of simplicity we adopt the monitored workload of the University web server presented in Section 5.3. So, the workload is the one depicted in Figure 5.2 and the corresponding Petri net model in Figure 5.4. Let us assume the following mean arrival request rates per minute in each state: 1800 for *peak*, 1300 for *working*, 100 for *night* and 500 for *weekend*. They are modeled by transitions of name λ_{state} in the Petri net. Rates of transitions that model state changes were explained in Section 5.3. Evaluating this Petri net in isolation (without considering the workflow Petri net), we obtain that its long term mean inter-arrival time is 943.4 requests per minute.

Characteristics of the processing resources We have supposed a set of identical servers and Round-robin technique to balance requests. We have followed classical techniques to create the SPN that models the load balancing technique, which inserted in between the workload and workflow SPN submodels. The characteristics of a server are:

1. Maximum power consumption, $W_{max} = 100W$.
2. Idle power consumption, $W_{standby} = 15W$.
3. Others power consumptions, $C_{startup} = C_{shudown} = 6000\ Joules$.
4. Frequencies range from 1600MHz to 3200MHz in steps of 266.6MHz. Thus, the set of frequencies is $\{50\%, 58.33\%, 66.66\%, 75\%, 83.33\%,$
$91.66\%, 100\%\}$.
5. Booting and shutdown times, $T_{startup} = T_{shutdown} = 1min$.

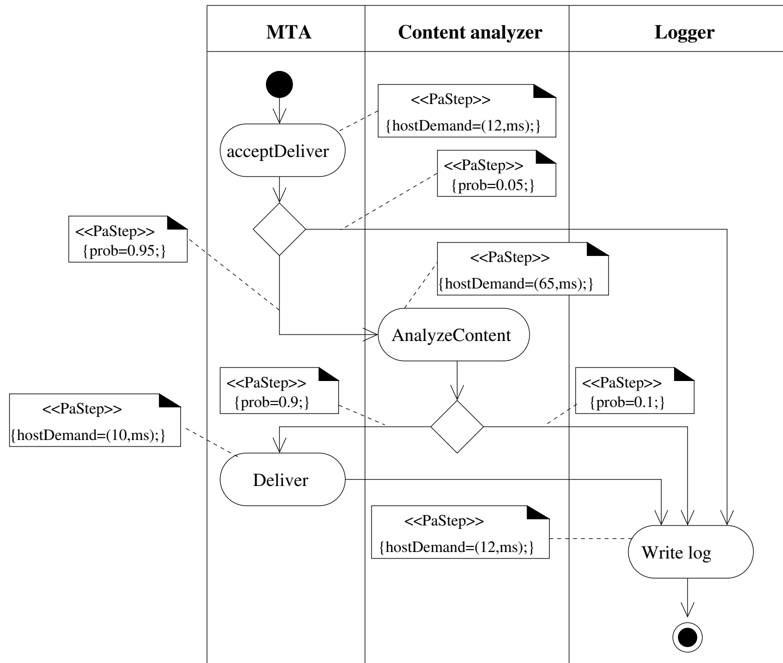


Figure 5.12: UML activity diagram of a relay server

Not energy-aware deployment We first conduct a study intended to devise the necessary amount of servers to cope with the performance requirement. This study does not heed about energy, so the servers are working at maximum performance and power consumption. The workflow in Figure 5.12 is translated into a SPN following the method in [LGMC04], let us call it SPN_{wkf}. The workload model is simplified and split to only consider *working* and *peak* periods, the worst cases. These nets together with the Round robin PN are attached to the SPN_{wkf}. As a result, we get two SPN we call $SPN_{wkf}^{working}$ and SPN_{wkf}^{peak}. We evaluate these nets for a different number of servers using the GreatSPN tool [Gre], and applying Little's law[4] we obtain the execution mean response times. We obtain that, if the platform consists of one or two servers, the system cannot satisfy required response time in *working* or *peak* states. Actually, system is neither able to manage the workload, and mean response times tend to infinite. However, for a three server platform, system satisfies the required performance in both *working* and *peak* states, since the obtained mean response times in this case are 0.16 and 0.73 seconds respectively. So, the system would be deployed in a three server platform. Servers power consumption using this solution would be $100W \cdot 3 = 300W$.

[4]The Little's law establishes that the average number of customers in the system is equal to the request arrival rate multiplied by the average time a customer spends in the system.

Energy-aware adaptation Previous not energy-aware study tells us that probably the three server solution is wasting energy since not all processing capacity is needed. Hence, we have to apply the energy-aware plan developed in Section 5.5 to find suitable configurations of servers for all the states in the system.

Firstly, we develop a table, as the one explained in Section 5.5, by evaluating the SPN that represents the workflow. This Table 5.2 embeds the different system configurations and shows for only one server its power consumption and the number of concurrent requests it can manage. Remember that under the assumption of "servers independence" we can obtain new rows for more servers just multiplying results in the first row.

	Operative frequencies (percentage)						
	50	58.33	66.66	75	83.33	91.66	100
W_{server}	25.6	31.8	40.1	50.8	64.1	80.4	100
Nrequests	10	12	14	15	17	19	21

Table 5.2: Power consumption and concurrent requests capacity

From this table the energy-aware plan is generated following indications in Section 5.5 and improved following the hysteresis in Subsection 5.5.2. We have generated three plans, considering different values for hysteresis step length, 1, 2 and 3. In the following, we refer to each plan as P_{H1}, P_{H2}, and P_{H3} respectively. The plans account for requests received in the last 2 seconds (*monitoredTimeSpan*). Figure 5.13 depicts P_{H2} as a graph. It shows that there are 11 different configurations and that the three server deployment is used when system load exceeds 22 requests.

We evaluated the generated plans and we obtained results for the steady state system execution:

- Mean time between consecutive booting processes are 33, 38.5 and 51 minutes for P_{H1}, P_{H2}, and P_{H3} respectively. It can be seen that the more step length of hysteresis, the less frequent booting processes are. Therefore, for each plan, expected mean power consumption of a server due to booting process is 2.86W, 2.6W and 1.96W for P_{H1}, P_{H2}, and P_{H3}, respectively, calculated as $\frac{6000J}{booting(P_{Hx})min \cdot 60seg/min}$.

- Percentage of time a server is turned on for each plan should be 78.3%, 79.4% and 81.2% . This has been directly acquired from mean number of tokens in `operativeServers` and `WaitForRequestsCompletion` places

- Percentage of time that the energy manager orders the infrastructure to work in each frequency is shown in Table 5.3.

Using all this information, we calculate the average power consumption of the system. Results are: 129.8W for P_{H1}, 140.6W for P_{H2} and 150.0W for P_{H3}. Therefore, our adaptation plans should make the system save 56%, 53.1% and 49.9% respectively w.r.t the non adaptive solution (calculated as $100 \cdot (300 - Power(P_{Hx}))/300$).

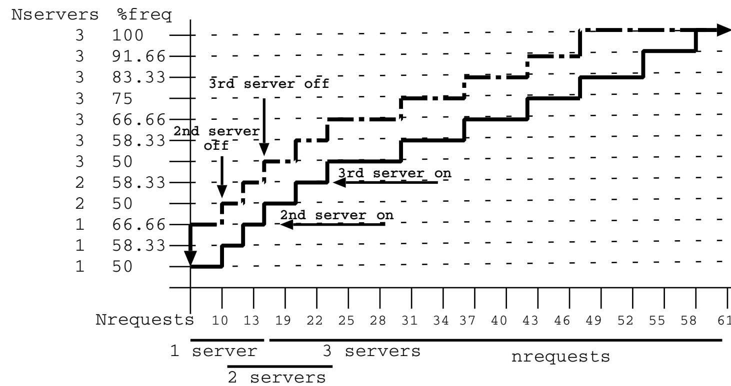


Figure 5.13: Graph for system reconfigurations

5.7 Experimenting with variable workload

The example illustrated in the previous section showed that the application of the proposed adaptation plan could lead to more than 50% of energy saving. The results obtained so far are tied to a given workload pattern. In this section, we analyze different types of workload, trying to understand if the results present a general validity or if they are related to specific situations. Hence, we expect to answer questions such as: “does it exist a workload state making the system unstable?” or “what could happen if the mean arrival rate of a workload falls just over the value where the adaptation plan suggests to increase the executing platform by one server?”. Answering these questions would increase or decrease the trust in the proposed approach and it could lead, for example, to the identification of critical workloads that require continuous system reconfigurations, thus deteriorating the energy consumption and servers wear and tear.

To this end we have performed several experiments, described below, whose goal was covering a wide range of workload rates to discover the existence of a possible critical workload state.

For the sake of system stability, while the state of the workload is not changing, the adaptations to activate servers should be close to zero since they take time (e.g., *booting* time). Indeed, in this case, high spike or deep valleys are not related to an increment or a decrement in the future workload, rather they are random events showing momentary workload variations. Therefore, a reconfiguration would entail to come back to the previous configuration in a very short time. In the worst case, it is possible to have a set of servers wasting time and energy turning on and off continuously, instead of processing requests.

To assess our approach, we have studied the behavior of the plan generated in Section 9.4 under several workload states, where each state has a different request arrival rate. Specifically, we have considered arrival rates from 50 to 2500 requests per minute in steps of 50 arrivals per minute. So, we have evaluated 50 kinds of workload rates. Furthermore, since

	Operative Frequencies (percentage)						
	50	58.33	66.66	75	83.33	91.6	100
$\%\, time\ P_{H1}$	19.8	13.6	19	18.3	17.4	8	3.7
$\%\, time\ P_{H2}$	20	9.2	11.4	18.3	22.7	12	6.4
$\%time\ P_{H3}$	20.9	13.6	0.8	9.5	27.3	18.2	9.7

Table 5.3: Percentage of time spent in each operative frequency

the hysteresis step length of the adaptation plan was conceived to reduce these reconfiguration rates, we have included in our experimentation also different lengths of the hysteresis step. In this way, it is possible to evaluate whether the hysteresis-based adaptation plan is effective to actually mitigate the amount of unnecessary reconfigurations.

For the completeness of the study, we have performed three different studies, using hysteresis step lengths ranging from 1 to 3. The obtained results are depicted in Figure 5.14. The graph shows the mean rate of unnecessary adaptations that modify the amount of active servers by varying the requests arrival rates. It can be seen that for an arrival rate of less than 100 requests per minute, the system is stable because none of the plans propose unnecessary reconfigurations. The same happens for arrival rates above 1350 requests per minute. In the former case, the system is stable using a 1-server configuration. In the latter, the system is stable using always the 3-servers configuration.

Between 950 and 1350 requests per minute, the adaptation with hysteresis step length equal to one proposes some unnecessary reconfigurations. Indeed, there are random valleys in the short term arrival rate that deceive the energy manager into changing to a 2-servers configuration, and then come back to the 3-servers one. The adaptation plans with hysteresis step length two and three, instead, are stable.

Between 250 and 500 requests per minute, it is possible to observe an increase of the rates of unnecessary reconfigurations of plans with hysteresis step of length equal to one and two.

Referring to the example of the previous section, this is the main reason of the observed difference between the value of P_{H1} and P_{H2} mean time between consecutive booting processes w.r.t. P_{H3}. Indeed, the mean arrival rate in *weekend* workload state falls in this range, which causes a certain instability in the system configuration. However, increasing the hysteresis step length to three, the reconfiguration rate decreased by a factor of 35.2% and of 24.5% respectively, mainly due to the avoidance of a number of unnecessary reconfigurations.

The adaptation plan with hysteresis step length equal to three also shows a peak in the unnecessary reconfiguration rates, that is shifted to the right w.r.t. peaks of plans with hysteresis step lengths one and two. However, as expected, this peak is much lower than the peak of previous plans, indicating that the increase in the hysteresis step length reduces the reconfiguration rate so increasing the system stability.

Experimenting with variable workload, we can conclude that it is possible finding workload states leading to unnecessary system reconfigurations, but the introduction of hysteresis helps in reducing the rate of reconfigurations. Specifically, adding hysteresis to the approach,

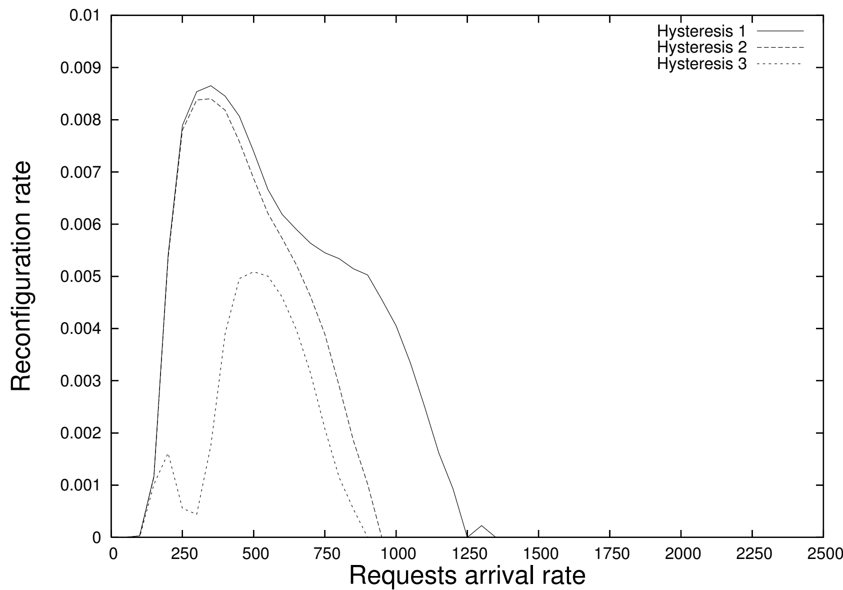


Figure 5.14: Reconfiguration rates for workload states with arrival rate $50r,\ r \in \{1..50\}$

we have experimented that it is possible to define suitable hysteresis step length values allowing the system to execute in a stable manner. We observed that increasing the length of the hysteresis step it is possible to reduce the mean power consumption and the reconfiguration rate. Finally, note that depending on the cost of servers and its wear and tear in each switching, the optimum value of hysteresis step length varies.

5.8 Conclusion

In the near future, the management of power consumption in open systems and computing infrastructures will necessarily become an unavoidable topic, self-adaptive frameworks have a lot to say at this regard. Starting from a framework (reference architecture) able to self-adapt a system to improve its performance, the proposal herein reported enhances the architecture to also deal with energy variables. The model-driven approach, transformation of UML models into a formal one in terms of Petri nets, bestows interesting analyses capabilities on the framework to carry out an off-line management.

Future work is a path plenty of challenges. For example, understanding how to apply virtual layers in the underlying computing infrastructure and how the architecture can manage them, servers will be able to feed different system components. Other not addressed topic refers the management of servers with heterogeneous capabilities (e.g., different frequencies). Also it is worth to investigate automatic generation of workload models from traces. In

particular, works on Markovian model estimation can be found in [CMCS12, CZS10].

5.9 Related work

In the last years, as outlined in [CdLG+09], the topic of reconfigurable and self-adaptive computing systems has been studied in several communities and from different perspectives. The *autonomic computing* framework is a notable example of a general approach to the design of such systems [KC03, HM08]. The work in this chapter lies in the area of models for self-adaptation of systems able to guarantee the fulfillment of performance requirements under variable workload and reducing energy consumption. Therefore, hereafter, we review works appearing in the literature dealing with (i) *dynamic variable workload*, (ii) *energy waste reduction* and (iii) *trade-off between energy consumption and performance*.

Workload Patterns for workload recognition and characterization have been studied in [SWHB06]. Differences between systems analysis depending on whether the considered workload is open, closed or partly open are explained, and difficulties to characterize the workload and difficulties to create and set up a suitable generator are discussed. For the problem we are dealing with, the more suitable workload types are open or partly-open. Moreover, we study systems under highly variable open workloads [WC03] with temporal dependencies. Besides, in this work we concentrate on the modeling of workload patterns that can be separated into phases, section 5.3 leveraged this aspect.

Energy wastes In [CD01], the authors outline a research agenda to reduce energy consumption in server clusters. The main proposal here is to improve server efficiency in terms of energy spent for each service request. The method focuses on reducing energy consumption by turning off the surplus of processing capacity when the current workload, which fluctuates, does not need it. With respect to this work, we manage the server frequency and the cubic relation with power consumption, which increases the energy saving with respect to only switching on and off servers.

Authors in [USC+08] have considered dynamic allocation of resources and deal with multi-tier applications. Although, they do not directly address the concept of energy savings, methods given in this paper can be applied for energy consumption reduction. In [BLM10] the management of energy consumption in data centers is studied through optimization problems taking into account: frequency scaling, severs booting times and hysteresis. We also assume these issues but we make slightly different assumptions to discover appropriate settings for each workload. For example, to accommodate an increased workload in the platform, in [BLM10] the plan could dictate switch on or off the machines. In our case, with increasing workload if you have to select a new configuration of hardware, we decided to not reduce the number of servers. Therefore, in this case we can reduce the number of reconfigurations.

Trade-off between energy consumption and performance This aspect has been largely investigated in hardware design, in network communities and in battery-powered devices. However, this investigation applied to hosting centers is much more recent. In [BR04, Ran10]

the importance of the problem of energy wastes is recognized. They treat the problem from different points of view, such as the consumption from hardware devices, operating system or software applications. They sum up previous efforts in the field, raise current problems and devise ways to reduce the energy consumption.

[EKR03] evaluates five strategies to save energy: two strategies manage processor frequency, another one switches on and off servers and the last two result from the combination of frequency and number of servers management. The authors study the performance degradation of applications with respect to the strategy used. In our work, we propose to generate an adaptation plan that uses the same techniques as in their latter strategy. Besides, we share the modeling of servers startup, shutdown and waiting for ongoing requests times. To predict execution demands of requests from each user our analyzable SPN models include more fine-grained information.

The goals of the work in [CDQ+05] are close to ours: to reduce costs while satisfying quality contracts. We share the techniques to save energy when the system is overdimensioned for the supported workload: switch off of the servers and modification of their frequency. Their optimization technique also considers the problem of wear and tear on servers when repeated on-off cycles are performed.They proposed methods based on queuing theory, feedback control and hybrid mechanisms, instead, we use SPN models in an architectural framework. We also differ because their proposal reconfigures the system just in predetermined time instants, however we do it as soon as a better configuration is recognized.

The authors in [AAA+06] propose a framework for hosting multi-service platforms that allows the management of reallocation of the correct amount of resources for each service while satisfying the performance requirements. The work in [CVP+08] extends the previous one by considering energy consumption constraints and situations where the system is under illegitimate users requests. Our work differs from the previous ones in the goals. While their main objective is to maximize company profits (they consider cases when providers pay penalties), our goal covers both the savings in energy consumption and continuous performance requirements satisfaction.

Mistral [JHJ+10] handles multiple distributed applications and large-scale infrastructures to optimize power consumption, performance and the transient costs of adaptations. As in our approach, Mistral reconfigures the system when variations in the monitored workload are appreciated, however they implement a workload predictor that estimates these workload variations, in our case the SPN model of the workload owns this knowledge. They present an algorithm, that can increase exponentially, to create a graph that represents the system configurations and adaptation actions, in our approach the reconfiguration plan is represented also by a SPN model. For the computation of applications response time, Mistral, as well as our approach, relies on formal models, in this case queuing networks instead of Petri nets. [HBK11] presents an approach to self-adaptive resource allocation in virtualized environments that cares for SLAs. Their adaptation algorithm differs from ours since it proceeds in two phases: a first one to allocate resources to meet SLAs and a later one to deallocate those not utilized. The approach is validated using standard benchmarks.

The approach in [KKH+09] implements and validates, using a benchmark, a dynamic resource provisioning framework for virtualized server environments. It also accounts for the switching costs of the machines. As in our approach, the excessive switching and the

variations in the workload intensity are taken into account. However, the approaches differ considerably. For example, they use a Kalman filter to estimate the number of future arrivals, while our approach allows accurate modeling using SPNs of multiple kinds and combinations of variable workload. The dynamics of the system are expressed using equations, however we use SPNs as a modeling paradigm.

Finally, [CJH+11] is an interesting work that develops a measurement-based approach as alternative to queuing models, which clearly differentiates it from our work. They also create a new set of metrics to predict runtime trade-offs between performance and energy use. Moreover, the alternative is extensively validated.

Chapter 6

Workload Modeling for Self-adaptive Software

As we have seen, software can be often embedded in dynamic contexts where it is subject to high variable, non-stable, and usually bursty workloads. A key requirement for a software system is to be able to self-react to workload changes by adapting its behavior dynamically, to ensure both the correct functionalities and extra functional requirements. Research on fitting variable workload traces into formal models had been carried out using Markovian modulated Poisson processes (MMPP). These works concentrated on modeling stable workload states, but accurate modeling of transient times still deserves attention since they are critical moments for the self-adaptation. In this chapter, after a detailed problem description, we build on research in the area of MMPP trace fitting and we propose a Petri net fine-grained model for modeling highly variable workloads that also accounts for transient times.

6.1 Motivation

Among the multiple sources of change that a self-adaptive software can face and multiple adaptation mechanisms, in this chapter we deal with changes in the workload and the adaptation of the processing resources allocated to the application task.

The workload, for some kind of systems, is far from being stable but it presents high variability and shows *burstiness*, i.e., irregular spikes of congestion. This is a fact for example in networked and service-based systems, but not only [MZR$^+$07]. If the workload model does not account for the existing *burstiness*, then the model analysis can lead to optimistic results; e.g., it declares a fair resource utilization and probability of congestion, while in the real setting they would not be guaranteed. In this chapter, we present our research on the modeling of workloads that show bursty periods.

Some formal methods that can model workloads considering the *burstiness* in the arrival rate are the Markov arrival processes (MAP) and a concrete subtype of them, the Markov modulated Poisson process (MMPP) [FMH93]. Research on workload and network traffic

fitting using MAPs and MMPPs have been already done and their results show an accurate modeling of the workload variability.

In particular, work on fitting MMPP and MAP parameters from workload traces with *burstiness* is very useful for the analysis of properties, such as performance or availability, of a wide range of systems. However, when we observe workload-aware self-adaptive systems carefully, we realize that their optimal configurations are different depending on the workload they are receiving. Moreover, differences between optimal configurations during the bursty periods and any other period can be huge. These systems should adapt (e.g., provisioning or release of resources) during *transient periods*, i.e., when the workload is becoming *bursty* and when the burst of arrivals is finishing. Usually, there is no need for this type of self-adaptive software to change its configuration during stable periods of workload, it should have been adequately provisioned before, in fact during these transient periods.

Therefore, to properly analyze the performance or availability of self-adaptive systems under bursty workloads, we need an accurate model of that workload. This model should include transient times, even when they correspond to a small percentage of the total time (the rates normal and burst can last for hours while the change between them lasts just some minutes). Otherwise, results from model-based system analysis can be far away from results of the real working system. The reason is that the system starts the adaptation when it anticipates the workload is close to be *bursty*. In this way, when the burst of requests arrive, the system is already in its optimal configuration. However, a system model whose workload does not care about transient times is not able to anticipate workload changes, and it will start its adaptations when the bursts of requests are already arriving. This can lead to too pessimistic performance and availability results from the model analysis.

In this chapter, we propose a model to take into account these transient periods. We build on the work done in [Gus91, CMCS12] for MMPP and MAP parameter fitting and we extend the generated models to be able to deal with self-adaptation. We start with the description of MAPs and MMPPs in Section 6.2, and we present the research on which we rely for MMPP parameter fitting from a workload trace. Section 6.3 explains the meaning of the *transient* time and proposes a model for its representation. In Section 6.4 we put together the MMPPs model and the new model for the transient time and we present the aggregated workload model. Using that aggregation of models, we are able to analyze more accurately the extra functional properties of the software. This is illustrated in Section 6.5 through an experimental analysis that shows the difference between considering or not the transient time in the workload model by evaluating the performance and availability requirements of a self-adaptive system. Sections 6.6 and 6.7 present the chapter conclusions and related works, respectively.

6.2 MAP's and MMPP's

Accurate characterization of real workload traces is a need to devise a proper workload model. For some kind of systems, e.g. networked ones, such characterization should capture the high variability of the requests as well as the fact that they can *burst* in on the system sometimes [MZR+07].

MMPPs are suitable to model variability and autocorrelation for event generation. An

MMPP is a stochastic process that has been extensively used to model event arrivals processes and network traffic [FMH93, Gus91, HL86], which is able to represent high variability and temporal dependencies in the arrivals. In an MMPP, the arrival rate at each moment is determined by the state of a continuous-time Markov chain (CTMC). So, when the chain is in state *i*, the arrival process is a Poisson process with rate λ_i. An MMPP with N states is defined by an NxN matrix Σ representing the CTMC and a vector Λ of N components representing the arrival rates in each state.

$$\Sigma = \begin{pmatrix} -\sigma_{11} & \sigma_{12} & ... & \sigma_{1N} \\ \sigma_{21} & -\sigma_{22} & ... & \sigma_{2N} \\ & & ... & \\ \sigma_{N1} & \sigma_{N2} & ... & -\sigma_{NN} \end{pmatrix}, \Lambda = (\lambda_1, ..., \lambda_N),$$

where $\forall\, i, j, \quad \sigma_{ij} \geq 0, \;\; \lambda_i \geq 0$ and $\forall\, i, \sum_{j:j\neq\, i} \sigma_{ij} = \sigma_{ii}$.

The research in this chapter considers MMPPs with two states. One of the states will represent the normal arrival rate (and we call it *normal*) and the other will represent the bursty arrival rate (and we call it *bursty*). A graphical representation of this two-state MMPP is given in Figure 6.1. λ_1, the *normal* arrival rate, and λ_2, the *bursty* arrival rate, are supposed to be much higher than transitions rates σ_{12} and σ_{21}.

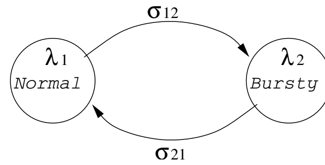


Figure 6.1: A two states MMPP

A two-state MAP, Figure 6.2, can be seen as a continuous time Markov chain of two states, and the active state defines the arrival rate. In the chain, there can be transitions associated with the arrival of an event (called *completion* transitions, λ_{ij}, darker in the figure) and transitions that are not associated with event arrival (called *background* transitions, σ_{ij}). Moreover, when the chain is in state i, it can also generate arrivals with rate λ_{ii} without changing its state, modeled as a self-transition, λ_{ii}.

Formally, a MAP can be defined by two squared matrices D0 and D1, where $D0_{ij}, i \neq j$ represents the *background* transition rates from state *i* to *j*, $D1_{ij}$ describes *completion* transition rates, and $D0_{ii} = -(\sum_{j:j\neq i} D0_{ij} + \sum_j D1_{ij})$. Thus, $Q = D0 + D1$ is the infinitesimal generator matrix of the chain.

An MMPP is a MAP that do not admit completion transitions that change the CTMC state, i.e., the elements not in the diagonal of $D1$ must be zero. Then, a two-state MMPP can be seen as a MAP whose matrices D0 and D1 are:

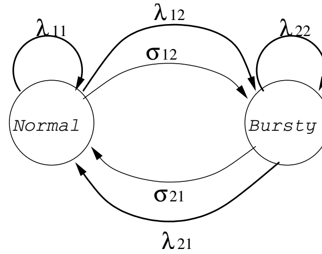


Figure 6.2: A two states MAP

$$D0 = \begin{pmatrix} -(\sigma_{12} + \lambda_1) & \sigma_{12} \\ \sigma_{21} & -(\sigma_{21} + \lambda_2) \end{pmatrix}, D1 = diag(\Lambda)$$

6.2.1 MMPP fitting from a workload trace

Finding the characterizing values of a trace To fit a real workload trace to a two-state MMPP we just need to set its four parameters: $\lambda_1, \lambda_2, \sigma_{12}, \sigma_{21}$. To this end, we will use four characterizing values from the workload trace.

The first value is the *index of dispersion for counts* (IDC) of the trace. The IDC is frequently used as an estimator of the *burstiness* in a trace. The higher IDC value is, the more *burstiness* the trace has. In [Gus91, HL86] it is calculated as

$$IDC_t = \frac{var(N_t)}{E(N_t)}$$

where N_t is the number of arrival in an interval of t time units. So, the IDC is the variance in the number of arrivals in t time units divided by the mean number of arrivals in t time units. Since we are interested in the index of dispersion of arrivals in the steady state, we calculate

$$\lim_{t \to +\infty} IDC_t$$

To calculate the IDC we use the algorithm presented in [MCCS08, CMCS12]. This algorithm is able to estimate the index of disperson $IDC_{t \to +\infty}$ of a single workload trace.

For the rest of the characterizing values we take advantage of the work in [CMCS12], that indeed fits workload traces to MAP caring about the burstiness. Besides the IDC, these values are: the mean inter-arrival time of requests (m), the 50th percentile (i.e, the median) and the 95th percentile. Since in that work the authors are characterizing the burstiness of service times, the burstiness happens for high values of these service times, then making important to know the value for which the 95% of service times are lower. However, we are dealing with inter-arrival times, and the burstiness happens when the values of inter-arrival times are low.

So, we prefer to know the value for which the 95% of times the inter-arrival time is higher than. For this reason, we use the 5th percentile instead of their 95th.

Experiment proposed As example of workload trace, we have used the monitored arrival times of requests to the FIFA 1998 World Cup site [Wor98]. This is the most complete example of workload trace we have been able to find. The timestamps are provided with granularity of one second and we have just used the requests that arrived to the Paris server region. Figure 6.3 shows the count of requests received by this region per minute. Since the workload was very low when the system was started and also the last days after the world cup, we have just concentrated in the middle days. We have used the arrivals of 34.7 consecutive days, then from minute 60,000 until minute 110,000. The arrivals in these 50,000 minutes have been considered in groups of 10 seconds and they are depicted in Figure 6.4. It is easy to see that the shape of the graph depicts a quite bursty workload. The selection of this time interval is not a restriction just to make the fitting algorithm work better but it exemplifies the kind of workloads we are really interested in. Since we are dealing with systems that are intended to continue working in the long term, we assume that the workload should not start and finish being low (as it happened to the World Cup website), but be always in the normal regime. So, we consider the first and last minutes as the system warm up and cool down, and we consider only the world cup days where the system was most used.

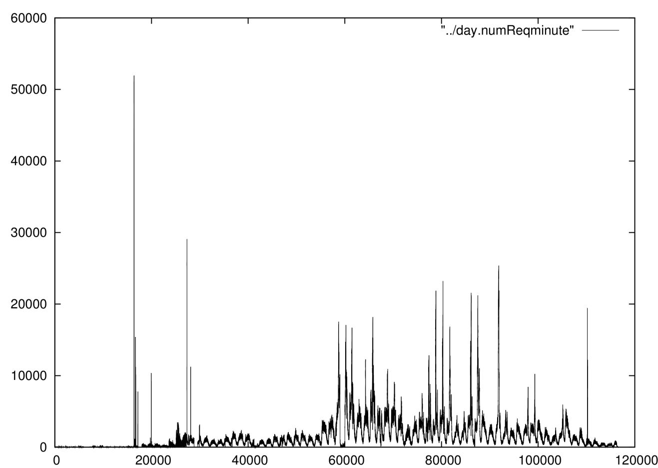


Figure 6.3: Requests per minute received in Paris region

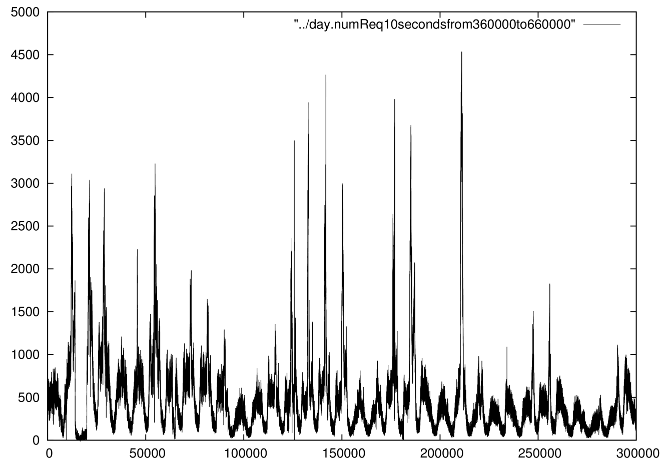


Figure 6.4: Requests every 10 seconds

Fitting MMPP parameters The characterizing values of the trace are the following. The number of requests that we have dealt with is 140,998,569. The mean inter-arrival time of requests is 0.021276 seconds (i.e, close to 47 requests per second), calculated as the number of received requests divided by $3 \cdot 10^6$ (the amount of seconds in 50,000 minutes). The median (percentile 50th) of the inter-arrival times is 0.0159744408 and the 5th percentile is 0.00367 (this is, the inter-arrival time of the 95% of requests was higher than this value). The IDC is 686,200, we admitted a tolerance of $1 \cdot 10^{-7}$ for its calculation using the algorithm in [CMCS12]. The amount of time that the algorithm considered approximate to infinite and for which the algorithm stopped was 45,140 seconds.

From these characterizing values, we fitted the MMPP. To fit the mean, 50th and 5th percentiles we have used the same equations as [CMCS12]. To fit the ICD, we have used the equation in [Gus91, HL86] that concretely deal with two-state MMPP parameters[1].

The results are:

$$\sigma_{11} = \sigma_{12} = 0.0000001314169$$

$$\sigma_{22} = \sigma_{21} = 0.0000273058047$$

$$\lambda_1 = 45.5395329586$$

$$\lambda_2 = 350.195877$$

[1]This equation is $IDC_{t\to+\infty} = 1 + \frac{2\sigma_{12}\sigma_{21}(\lambda_1-\lambda_2)^2}{(\sigma_{12}+\sigma_{21})^2(\lambda_1\sigma_{21}+\lambda_2\sigma_{12})}$

As expected, we can see that the mean sojourn time in each state, $\sigma_{12}^{-1}, \sigma_{21}^{-1}$, is orders of magnitude higher than the mean requests inter-arrival times, $\lambda_1^{-1}, \lambda_2^{-1}$.

6.2.2 GSPN workload model

An accurate workload model with burstiness, as the one proposed by the MMPP, is necessary for the eventual analysis of systems that execute under such conditions.

GSPNs [AMBC+95] are broadly used to model the behavior and workload of systems and also as analyzable models to predict properties of software systems. GSPNs have been used to analyze some properties of self-adaptive software systems, such as performance and energy, as it has been shown in Chapters 3 and 5. Since our workload model should represent the injection of requests in the system in the same language as the behavioral system model, we pursue the proposed MMPP workload model in terms of GSPN.

Since both GSPNs and MMPPs represent markovian processes, we can get a GSPN with the same behavior as the MMPP in a quite straightforward manner. This GSPN, the one in Fig. 6.5 representing the two state MMPP in Fig. 6.1, has as many places as states the MMPP, in this case P_1 and P_2 (for *normal* and *bursty*, respectively). Another place, $P_{arrivals}$, will mean the injection of requests in the system, i.e. injection of tokens in the GSPN that represents the behavior of the self-adaptive system. The time transitions T_{12} and T_{21} represent the MMPP change of state, then their firing rates are σ_{12} and σ_{21} obviously. The last two transitions, $T_{arrival1}$ and $T_{arrival2}$, represent the arrival rates in the MMPP, therefore their firing rates are λ_1 and λ_2 and they feed the $P_{arrivals}$ place.

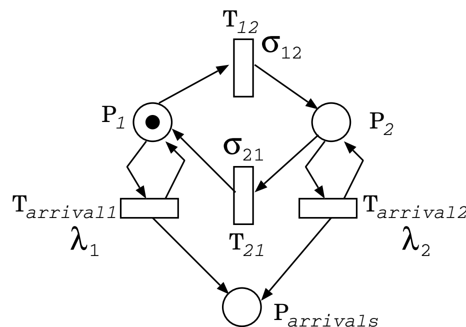


Figure 6.5: GSPN for the two states MMPP

6.3 Modeling transient time between states

6.3.1 Problem statement

As declared in the beginning of the chapter, a self-adaptive system needs some time to perform corrective actions (e.g., provisioning or release of resources) to fit into the new execution context. In systems whose adaptations depend on workload variations, such adaptations should happen when the system changes from *normal* to *bursty* or vice versa, i.e., the system adapts to the environment during the transient times between states.

When looking at the real workload trace in Figure 6.6 we observe that such transient time, although fast, is not immediate, it lasts for around 41.6 minutes, starting around 850 and ending around 1100 ($\frac{1100-850}{6} = 41.6$). The figure shows a period of 250 minutes which corresponds to the zoom in the range from 209,500 to 211,000 in Figure 6.4. The transient time is assumed to be fast w.r.t. the mean sojourn time in each stable state that last for many hours. Our workload model should reflect the transient time accurately since in this period the self-adaptive system:

- perceives that the workload is leaving the normal state and the burst of arrivals are near to arrive, and
- performs its adaptations to change its configuration to a new one able to withstand the burst of requests.

In a two-state MMPP the transient time is not modeled as we can see in Figure 6.7. This figure represents a workload trace generated by the fitted MMPP in Section 6.2 and we observe that the change from *normal* state (arrival rate around 455 requests each 10 seconds) to *bursty* state (around 3500 requests during 10 seconds) is abrupt, no transient time is perceived.

6.3.2 Setting parameters of workload model

We pursue a GSPN to model transient times in the real workload trace, i.e., the increments and decrements in the arrival rates of the requests. The zones of increment or decrement can be characterized by three parameters:

- the well-known λ_1 and λ_2,
- the amplitude of the zone, we call it mt_{inc} or mt_{dec}, they are measured in seconds, and they represent the mean amount of time that the workload is increasing from *normal* state to *bursty* state or decreasing from *bursty* to *normal* respectively,
- and additionally, from these parameters we can also calculate the acceleration of the curve in the zone, mr_{inc} or mr_{dec}, in $requests \cdot seconds^{-2}$.

In the following we describe how these parameters can be obtained from a real workload trace. Algorithm 6.1 shows the case of the calculation of the mean amount of time that the workload is increasing.

First (line 1 in Algorithm 6.1), we apply the technique presented in Section 6.2.1 to get λ_1 and λ_2.

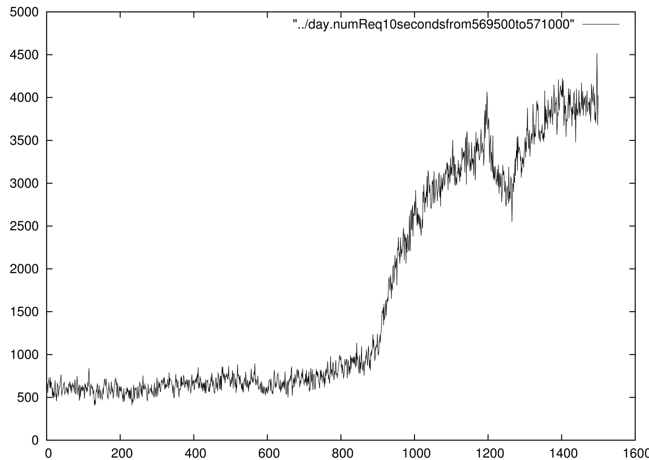


Figure 6.6: Real workload trace: focus on the increment

Second (line 2), we go all over the counts[2] in the workload trace. Let us call $count_j$ the number of requests received by the count in position j. We find each j such that $count_{j-1} < \lambda_1 \leq count_j$, we call it $candidate_j$. That is, the candidates are the counts where the arrival rate has changed from being under the mean for the *normal* state to be over the mean.

Third (lines 3..9), for the first $candidate_j$ we find the first $k,\ k > j$, such that $(\lambda_1 > count_k) \vee (count_k > \lambda_2)$.

- If the first condition holds, we can discard $candidate_j$ since it means that the workload is not incrementing, but it had just exceeded the mean for a while and it has returned under the mean again (this is the usual behavior when the workload is in a stable state).

- If the second condition holds, we keep $candidate_j$ since we could have found a period of increment in the workload from *normal* to *bursty* arrival rates, this period is $[j, k]$.

Fourth (lines 10..17), for each $[j, k]$ period we have to discover whether it can be considered as a real workload increment or not. We assume that a real increment happens when the counts between $[j, k]$ increase constantly in a *coarse-grained* view of the workload.

As *coarse-grained* we mean that we zoom-out the counts in order to mask the short-term variability. To create the *coarse-grained* view, we reduce the $k - j$ monitored counts to N values, where each $N_n, 0 \leq n < N < (k - j)$, counts the number of arrivals in a period of length L. L is a choice and represents how much coarse will be the study. A too low L will not avoid the short-term variability (and then we will not realize that the workload

[2]Remember that a count means the number of requests received in 10 seconds.

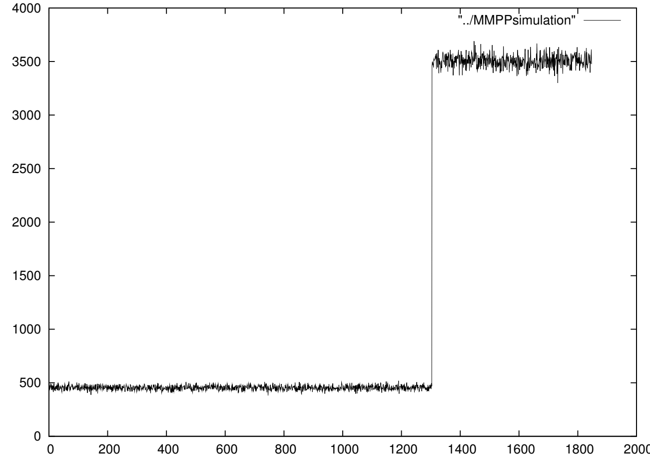


Figure 6.7: Workload trace modeled by the MMPP

is truly increasing), and a too big L will recognize as periods of constant increment some that should not be. Once L value is chosen, N is calculated as the largest value for which $N \cdot L \leq (k-j)$, i.e, $(k-j) < (N+1) \cdot L$. If $N \cdot L \neq (k-j)$, we obviate in the study the last values $k-j-N \cdot L$ of the interval, that is, the interval to work changes from $[j,k]$ to $[j, j+N \cdot L]$. Now, we sum up in N_n the values of each group of L counts. Therefore, $N_n = \sum_{i=0}^{L-1}(count_{n \cdot L+j+i})$.

To finish the fourth step, we decide that $[j,k]$ is a real constant increment. Ideally, a constant increment happens if the number of counts N_n are increasing values, i.e, if $\forall\ n \in \{1..N-1\},\ N_{n-1} < N_n$. However, we found that in every increment interval in the trace, there is at least one unexpected count of arrivals that is very different from its neighbor counts (too less or too much) that are also visible in the coarse-grained view. This unexpected count prevents satisfying the *for all* in the previous condition. To solve it, we add a percentage of tolerance $tol \in \mathbb{R},\ 0 \leq tol \leq 1$. Then, the amount of counts $N_n,\ n \in \{1..N-1\}$ that must satisfy the condition $N_{n-1} < N_n$ is reduced from $N-1$ (i.e, all counts) to $(1-tol)(N-1)$.

Fifth, if it has been decided that the interval represents a constant increment in the coarse-grained view, we get a parameter to characterize the interval $[j,k]$: the amplitude $t_{inc} = k-j$. Besides, we can derive more parameters from the interval such as the acceleration r_{inc} in the request arrival, calculated from λ_1, λ_2 and t_{inc} as $r_{inc} = \frac{\lambda_2 - \lambda_1}{t_{inc}}$.

Sixth, we repeat the third, fourth and fifth steps for all $candidate_j$.

Finally, using the discovered t_{inc} and derived r_{inc} in each iteration, we get the values of mt_{inc} and mr_{inc} as the mean of them (lines 18..23).

We perform the same steps to discover the periods of time where the workload is decreasing. Using these periods, we will obtain mt_{dec} and mr_{dec}.

Algorithm 6.1 Parameter estimation

Require: Workload trace with the *count* of arrivals
Ensure: mt_{inc}
1: (λ_1,λ_2) $\leftarrow$ MMPPfitting($count$);
2: $candidates \leftarrow$ findCandidates($count$, λ_1);
3: $intervals \leftarrow \emptyset$;
4: **for all** $candidate \in candidates$ **do**
5: $k \leftarrow$ getFirstCrossingValue($count$,$candidate$,λ_1,λ_2);
6: **if** $count_k$¿λ_2 **then**
7: $intervals \leftarrow$ addInterval($intervals$, [$candidate, k$]);
8: **end if**
9: **end for**
10: $L \leftarrow$ chooseL(); $tol \leftarrow$ chooseTol();
11: **for all** $interval \in intervals$ **do**
12: $N \leftarrow$ calculateN(L, $interval$);
13: $subtrace \leftarrow$ makeCoarse($trace$, $interval$, N);
14: **if not** isContinuousIncrement($subtrace, tol$) **then**
15: $intervals \leftarrow$ discardInterval($intervals, interval$);
16: **end if**
17: **end for**
18: $numberOfIntervals \leftarrow 0$; $incrTime \leftarrow 0$;
19: **for all** $interval \in intervals$ **do**
20: $incrTime \leftarrow incrTime + interval$.amplitude;
21: $numberOfIntervals \leftarrow numberOfIntervals$+1;
22: **end for**
23: $mt_{inc} \leftarrow \frac{incrTime}{numberOfIntervals}$;
24: **return** mt_{inc}

Note that mr_{inc} and mr_{dec} are real positive values ($\mathbb{R}^+$). So, we are assuming a constant acceleration and deceleration in the workload during transient time. On the one hand, this linear increment in the arrival rate is more accurate than the previously assumed immediate increment. Besides, the linearity in the increment/decrement corresponds to the long-term view of the increment, since we are still modeling the variability in the short-term. On the other hand, we are approximating to be linear any workload increment/decrement between states. Other possible representations of the workload increment/decrement are possible, but the identification of the kind of increment in the coarse-grained view is more complicated since we would also come into the field of curve fitting.

6.3.3 GSPN model for the transient time

The GSPNs in Figure 6.8 (a) and (b) model the transient time from *normal* to *burst* (increment in the arrival rate of requests) and from *burst* to *normal* (decrement in the arrival rate of requests), respectively.

GSPN model: from normal to burst A key point is that, during the transient period, the arrival of requests are modeled as tokens created in place $P_{arrivals}$ at a variable rate. This rate will be $\lambda_1 + \lambda_{inc} \cdot \#P_{inc2}$ since transitions $T_{arrival1'}$ and $T_{arrivalInc}$ provide the tokens[3]. The former transition generates the workload of the *normal* state, λ_1, while the latter transition generates the increment of requests[4].

A token in P_{inc1} means that the system enters in the transient state so leaving the *normal* one. Then, every σ_{inc} units of time a new token is set in P_{inc2} to precisely generate the increment of requests. When the number of tokens in P_{inc2} is w_1, it means that the transient time has completed and the system enters in the *bursty* state P_2 by firing transition t_{inc2}.

Although not yet observed in the figure, transition t_{12} will fire when the normal arrival rate of request in the system has finished, hence to start this transient period.

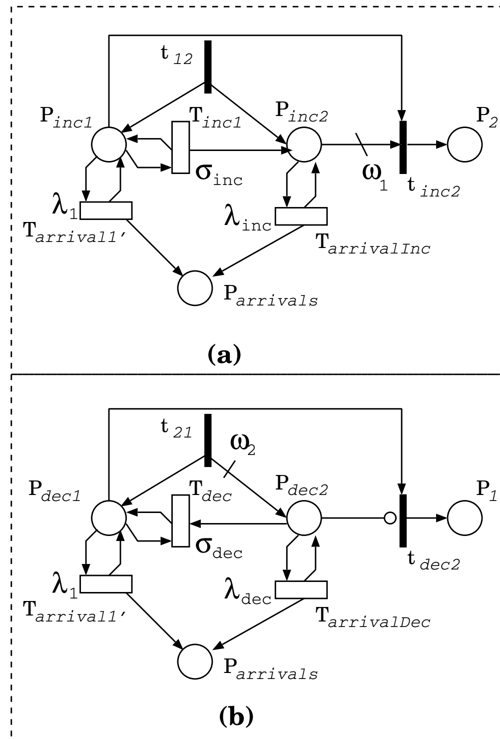


Figure 6.8: GSPNs: (a)increasing and (b)decreasing arrival rates of requests

[3] $\#P_i$ is the number of tokens in place P_i.

[4] It is worth noting that we are considering *infinite server* semantic for all transitions

Fitting GSPN parameters We use the four parameters computed in the previous subsection, λ_1, λ_2, mr_{inc} and mt_{inc}, to set the parameters of the GSPN, $w_1, \lambda_{inc}, \sigma_{inc}$.

First, note that the modeling of transient times increases the state space for the analysis. Fortunately, we can decide the amount of increment we allow. For the transient time that models the change from *normal* to *bursty*, the state space grows linearly with parameter $w_1 \geq 1$, and we can freely decide its value. The rationale is that w_1 corresponds with the amount of token variability in P_{inc2}. Then, observe that P_{inc2} can have tangible markings in the interval $[0, w_1 - 1]$, while markings where $\#P_{inc2} = w_1$ are vanishing and do not affect for the state space analysis. This allowed variability entails that we model $w_1 - 1$ increments in the workload between λ_1 and λ_2. So, each token will increase the arrival rate in $inc = \frac{\lambda_2 - \lambda_1}{w_1}$ units. This inc is the value of λ_{inc}. Finally, we calculate how fast are created the tokens in P_{inc2}, this is, we calculate σ_{inc} of T_{inc1}. This transition fires $w_1 - 1$ times for each transient period, and it should fire in mt_{inc} time units. So its firing rate $\sigma_{inc} = \frac{w_1 - 1}{mt_{inc}}$.

Now it can be easily seen that we preserve the short term variability in the workload increment since the arrival rate is still based on stochastic processes exponentially distributed.

GSPN model: from burst to normal The differences between this model, in Figure 6.8 (b), and the previous one are:

- t_{21} starts this transient state by setting w_2 tokens in P_{dec2}. Again, w_2 represents the amount of complexity we can afford to model the decrementing period in the workload. The size of the state space will be $w_2 + 1$ times the state space of the workload model without decrementing period.

- Tokens in P_{dec2} decrease at rate σ_{dec}, when P_{dec2} is empty the system enters in the *normal* state P_2. The σ_{dec} firing rate is $\frac{w_2}{mt_{dec}}$.

- The transient state generates requests at rate

$$\lambda_1 + \lambda_{dec} \cdot \#P_{dec2}.$$

  Where λ_{dec} is $\frac{\lambda_2 - \lambda_1}{w_1}$.

6.4 Comprehensive workload model

So far we have proposed GSPN models separately, one model for the two characteristic states, in Figure 6.5, and two for the transient times, the increment of the workload in Figure 6.8(a) and the decrement in Figure 6.8(b). Our challenge now is to merge these three GSPN models to get a single one that cares for burstiness and transient times as required by self-adaptive systems.

Before merging the GSPNs we need to slightly modify the net of Figure 6.5: we remove the arc from T_{12} to P_2 and the arc from T_{21} to P_1. The rationale behind this modification is that we want to avoid the immediate change between *normal*, P_1, and *burst*, P_2, and vice versa.

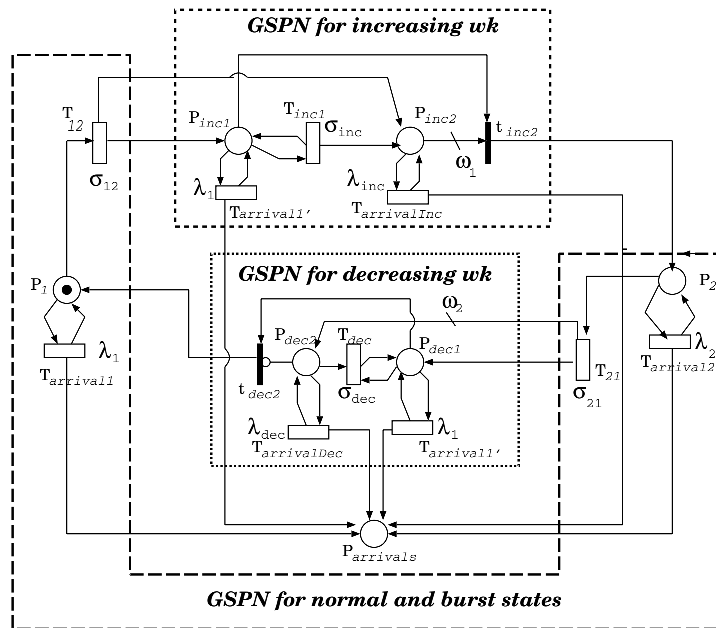


Figure 6.9: Complete workload model

The resulting GSPN is the one in Figure 6.9. We have used the composition operator for GSPNs formally defined in [DF96]. The essence of the operator is easy to understand, it overlaps the transitions (places) with the same name. For example, the place $P_{arrivals}$ appears in the three nets, however in the resulting net it appears only once, having as input arcs all the input arcs of the three original places.

The expert reader can argue that the GSPN in Figure 6.9 can be equivalent to a M-state MMPP where $M = 1 + w_1 + w_2$. In that case, the parameters of that M-state MMPP with the same characteristics as our workload model would be:

$$\Sigma = \begin{pmatrix} -\sigma_{11} & \sigma_{12} & 0 & ... & & & & 0 \\ 0 & -\sigma_{inc} & \sigma_{inc} & 0 & ... & & & 0 \\ & & & & & & & \\ & ... & & & ... & & ... & \\ & & & & & & & \\ 0... & 0 & -\sigma_{22} & \sigma_{22} & 0 & ... & & 0 \\ 0... & & 0 & -\sigma_{dec} & \sigma_{dec} & 0 & ... & 0 \\ & & & & & & & \\ & ... & & & ... & & ... & \\ & & & & & & & \\ 0... & & & & & 0 & -\sigma_{dec} & \sigma_{dec} \\ \sigma_{dec} & 0... & & & & & 0 & -\sigma_{dec} \end{pmatrix}$$

$$\Lambda = (\lambda_1, \lambda_1 + \lambda_{inc}, ..., \lambda_1 + (w_1 - 1)\lambda_{inc}, \lambda_2, \lambda_1 + w_2\lambda_{dec}, ..., \lambda_1 + \lambda_{dec})$$

Then, a question arise: could that M-state MMPP be directly obtained from the workload trace using the technique presented in Section 6.2 for two-state MMPP?

The answer is yes. Nevertheless, there are some restrictive challenges to obtain the characterizing values of an M-state MMPP. These are: the algorithm to fit parameters of an M-state MMPP is much more time consuming and the estimation of its parameters are much more prone to inaccuracies. Moreover the current techniques to fit MMPP parameters do not directly deal with our problem (gaining accuracy in the transient times models).

6.5 Experimental analysis

In this section, we illustrate the results obtained in our experimentation. To this end we have considered a very simple system with different workload models: first MMPPs and second our GSPN model, that includes the transient time between workload states. A third experiment is used as a benchmark for comparing the accuracy of the obtained results, it is a system simulation having a real workload trace, the one in Figure 6.4.

The system we use in this experimentation is a very simple software made of only one activity that requires on average 3ms of processing time [5]. There is a single processor executing a maximum of ten concurrent requests, queueing and serving them following a FIFO policy. Requests above ten are rejected.

We assume that requirements to architect the system are:

- R1- availability: *at least 99% of requests must be served, and*
- R2- performance: *the mean response time should be lower than 1 second*.

[5] To be able to compare approaches without including more variables that can distort results, we assume that the mentioned processing time is exponentially distributed with mean 3ms

Note that the response time is not a critical requirement, since the maximum length of queue of requests to be served is nine, and they are served in a mean of 3ms. On the contrary, requirement R1 is the critical one.

When we analyzed the system considering a workload model without burstiness (i.e., taking into account the mean inter-arrival time derived from the real trace), the requirements were satisfied. On the contrary, when taking into account the arrival in bursts, the analysis of the system showed that R1 cannot be guaranteed. A possible solution passes through the addition of a second processing resource. Now, having two processing resources, the system is able to satisfy both R1 and R2 also during bursty periods. However, the second processing resource has been added just to allow the requirements satisfaction during the periods of burstiness, which represents the worst-case scenario for the system. So, during the normal arrival rate periods, there is a waste of resources.

We can use the model proposed in Section 6.3 to take into account the workload variability. To this end, we consider a system enhanced with a monitoring component. The monitor is a passive observer that measures the system workload. Then, the monitor notifies to a separate component, which acts as a controller, when the workload is changing and when to add a second processing resource. In the same way, it also decides to switch off one of the processing resources when the workload decreases. So, the system deployment is no longer static but it is dynamically adaptable. It can be seen that we are applying architectural concepts, the monitor and controller are acting as the *sensor* and *adaptation manager* of the architecture presented in Chapter 2.

We have set the following parameters for the self-adaptive system:

- The maximum arrival rate of requests that can be served by only one processing resource is the 80% of its maximum capacity. In other words, the controller decides to add a new processing resource when the workload goes above $\frac{1}{3ms} \cdot 0.8 \approx 266$ requests per second.

- The maximum arrival rate of requests that can be served using both processing resources is 40 requests per second. When the workload rate is under this value, the second processing resource is shut down.

- Booting and shutting down times of the processing resources is one minute.

In the following, we explain the set-up of each experiment and the obtained results. After, we compare and discuss results.

MMPP workload model As MMPP workload we used the one already calculated in Section 6.2. We composed the MMPP model, in GSPN terms, with the GSPN that models the behavior of the described system. We analyzed the resulting GSPN and obtained the following results:

The percentage of requests rejected is 1.43%, so the availability is 98.56%; and the mean response time is 5.3ms.

Then, R2 is satisfied while R1 cannot be guaranteed.

MMPP with transient times workload model Using the MMPP parameters already calculated in Section 6.2 we applied the process described in Section 6.3 to identify in the trace in Figure 6.4 periods of coarse-grained-constantly-increasing workload.

The parameters L and tol have been set to 5 minutes and to 0.2, respectively. Then, the workload parameters mt_{inc}, mr_{inc}, mt_{dec} and mr_{dec} are:

$$mt_{inc} = 5192s \quad mr_{inc} = 0.58requests \cdot s^{-2}$$
$$mt_{dec} = 3770s \quad mr_{dec} = -0.8081requests \cdot s^{-2}$$

Following the procedure described in Section 6.3 we defined the structure of the GSPN models for the transient times. We then used the previous results as parameters of these GSPN models.

To complete the model definition, we decided the amount of affordable increment in the state space as $w_1 = 10, w_2 = 9$. Using these values, the remaining GSPNs parameters $w_1, \lambda_{inc}, \sigma_{inc}, w_2, \lambda_{dec}$ and σ_{dec} have been derived.

The GSPN modeling the workload has been obtained as described in Section 6.4 by composing the MMPP part with the GSPN derived for the transient times. Next, we composed the GSPN workload model with the GSPN that represents the behavior of the system. We analyzed this GSPN and we obtained the following results:

The percentage of requests rejected is 0.56%, so the availability is 99.44%; and the mean response time is 5ms.

Hence, R1 and R2 are satisfied.

Real workload trace execution For validation purpose, we have implemented a simulator of the system described in the example. We run the simulator and we injected the requests following the real workload trace.

We have obtained the following results:

The percentage of requests rejected is 0.05%, so the availability is 99.95%; and the mean response time is 3.7ms.

With the simulation and the real workload both requirements are satisfied.

	MMPP	**MMPP with transient times**	**Real trace**
Availability	98.56%	99.44%	99.95%
Performance	5.3ms	5ms	3.7ms

Table 6.1: Evaluation results with different input workload

6.5.1 Results discussion

Looking at Table 6.1 we can observe that the results obtained with both the MMPP model and MMPP with explicit transient time workload model are pessimistic with respect to the

real ones. Indeed, the analysis of the models produced results showing lower availability and higher average response time with respect to the results obtained by the system simulation using the real workload trace. However, the results obtained with the MMPP including the transient time model are better than the ones obtained with the simple MMPP and closer to the system simulation results.

Actually, in this simple example we can see that the expected rejection probability of requests from model analysis with MMPP is $\frac{1.43}{0.05} = 28.6$ times higher than the calculated by simulating the real trace. Adding the transient times to the MMPP model, we have reduced this error to be $\frac{0.56}{0.05} = 11.2$ times higher; so, we have brought the result a 60% closer to the real one. Besides, the conclusion from the analysis of the model with MMPP workload would be that the proposed adaptive solution for the system does not satisfy the availability requirement. This decision would be wrong because the actual system satisfies it.

Regarding the mean response time, adding the transient times to the MMPP model, we have just reduced the error of the results from being 1.43 times the real ones to be 1.35 times.

Note that, although all the experiments regarding requirement R1 seem to produce very similar results, this is not the case since availability is used to be measured as the "number of nines". In other words, if we compare a system with 99% of availability and another one with 99.9%, the latter is not just 0.9% more available than the first one but it is ten times more available. In our experiments, the availability obtained with the MMPP workload model without transient times resulted 28.6 times lower than the availability of the system with the real workload. Adding the transient time between states to the workload model we have been able to reduce the error of around the 60%, of course this is still not enough to guarantee results very close to the real ones.

6.6 Conclusion

Modern techniques to model high variable workloads and burstiness are based on markovian models such as Markov arrival processes and Markov-modulated Poisson processes. They offer a powerful theory to model workload. In this chapter, we have identified a need for the accurate workload modeling for self-adaptive systems. This need refers to the modeling of the transient time between workload states in the presence of *burstiness*.

This transient time is not modeled in MMPPs, because they focus on modeling stable workload states. Although these transient times may not be important for static systems or workloads without *burstiness* characteristic, they are crucial when analyzing bursty workload-aware self-adaptive systems. To solve this challenge we have built on previous results on MMPP fitting and we have proposed a model based on Petri net taking into account the arrivals during the transient time between states. The obtained model has then been integrated in a Petri net describing the MMPP, so allowing a more complete representation of the workload.

A first experimentation comparing the results obtained with the proposed model and the classical MMPP models tested against a real trace workload, showed an increment in the analysis accuracy when transient times are taken into account.

Besides, from our experimentation we have shown that we have reduced the errors in the analysis results; although there is still a gap between model analysis results and real

simulation ones.

A direction that deserves further investigation is the representation of the workload transient times when there are more than two stable states. In these cases, the MMPP that models the stable states has more than two states. Since the addition of the transient time models increases the state space of the model to analyze, it may not be appropriate to represent the incrementing and decrementing transient times between any two states. Contrarily, we should search which state transitions deserve attention to model their transient times and which ones do not deserve it.

6.7 Related work

The parameter fitting of Markovian models such as MMPPs and MAPs is an extensive research field. For example, works [HL86, HT02, OD09, Gus91, Ryd96, CZS10, CMCS12] propose MAP and MMPP parameter fitting techiques starting from traffic traces. Some of these fitting works also deal with the modeling of burstiness characteristic and use the index of dispersion as burstiness estimator.

This chapter builds on the results obtained in [Gus91, CMCS12] to choose the estimators of the workload trace and fit a two-sate MMPP that models the same characteristics as the workload trace for these estimators. However, to the best of our knowledge, our work is the first one modeling the transient times between workload states and using them when evaluating workload-aware self-adaptive systems.

Chapter 7

Measuring and Correlating System Adaptability

Previous chapters dealt with system properties that can be quantified: performance and energy consumption. However, adaptability property was considered as a purely qualitative property: a system can be *non-adaptive* or *self-adaptive*; and usually only these two states are distinguished. Once a system is classified as *self-adaptive* there is not any manner to compare its adaptability with the adaptability of other systems; it is not possible to rank them attending to adaptability. The goal of our research in this chapter is to overcome this limitation and to enable comparison of software systems regarding their adaptability.

7.1 Problem statement

Adaptability is a property that can be evaluated at different stages of the application development, e.g., at architectural level. Indeed, in recent decades, software architecture has emerged as an appropriate level for dealing with software behavior and qualities [CKK01, SW02b, BCK05] and several efforts have been devoted to the definition of methods and tools able to facilitate the actual system development and to evaluate quality at the architectural level (see, for example, [BCK05, BDIS04, DN02, SW02b]).

In this chapter, we propose the definition of metrics allowing the description and the evaluation of the system adaptability at the architectural level. We believe that the existence of metrics able to quantify (even if in a simple way) the system adaptability could provide a key capability for the development of systems that can adapt when necessary. These metrics can allow the comparison of different possible architectures with respect to their potential for adaptation. Besides, we argue the importance of defining a relationship between the adaptability and the quality properties of the system. Our metrics can then be used to drive the system adaptability in order to meet the overall QoS requirements. Moreover, the metric definition can be seen as a first step towards future research on formal approaches for the evaluation of system adaptation.

The idea of defining metrics for quantifying software adaptability is not new. This idea was first proposed in [SC01] and then refined in [RM09]. Our approach is built on these previous works and improve them by presenting a wider set of metrics and by defining the aforementioned relationship between them and the quality requirements.

Indeed, adaptability can influence software qualities such as performance, reliability or maintainability and in the worst case, improving the adaptability of a system could decrease other qualities. Finding the best balance between different, possibly conflicting quality requirements that a system has to meet and its adaptability is an ambitious and challenging goal that this research pursues. As a first step towards this goal, in this chapter we present a method for evaluating the relationships between the system adaptability and two qualities, concretely availability and cost.

Far from being "a solution for every situation" these metrics and relationships can enable software architects to discover suitable architectures leading to quality requirement satisfaction. The obtained adaptability values of each requirement can then be combined to evaluate the various trade-offs and decide whether there exists an architecture that fulfills all client requirements. The required input to perform these tasks concerns the execution context, in terms of existing or planned software resources, and their quality attributes.

The evaluation of architecture alternatives is executed when the software architect should take into account, for example, the introduction of new clients, or changes either in already known clients requirements or in the context (in terms of its existing elements or the quality attributes of an element) possibly preventing the clients satisfaction. After the generation of the knowledge about a suitable set of architecture solutions that fulfill the requirements, software architects will choose one solution based either on client desires or on any own-system preference.

The rest of the chapter is organized as follows: Section 7.2 describes the proposed metrics for quantifying the adaptability of a software. Section 7.3 investigates relationships between the adaptability metrics and extra functional requirements. We apply this approach to a simple example in Section 7.4. Section 7.5 presents a trade-off analysis among different extra functional requirements. Sections 7.6 and 7.7 complete the chapter by explaining our conclusions and describing the related works.

7.2 Architectural adaptability quantification

This section presents the definition of some metrics for quantifying the adaptability of software systems at architectural level. The goal of these metrics is to give a means for evaluating the potential of the system to adapt rather than a description of how the the system will adapt.

7.2.1 Architectural assumptions

For the metrics definition we refer to an architectural description formed by components (hereafter denoted as *software unit*) and connectors. We will use the UML component diagram to represent them, see Figure 7.1. By *software unit* we mean for example components, in the context of component-based software engineering, or Internet services, probably provided by third-parties, in the context of SOA. The connectors in our description represent the

relationships between the different software units and indicate, for example, that a software unit requiring a service is connected to other software units offering the service. By *software unit of interest* we mean a software unit chosen by the architect, among all available in the market, as candidate to make up the system.

In the example, the system offers functionality *f1* exclusively through *SU11*, which in turn needs *f2* (offered by *SU21* and *SU22*) and *f3* (by *SU31*, *SU32* and *SU33*). Therefore, SU_1= {*SU11*}, SU_2= {*SU21, SU22*} and SU_3= {*SU31, SU32, SU33*}. The units of interest are in grey, E_1= {*SU11*}, E_2= {*SU21*} and E_3= {*SU31,SU32*}, then the only adaptable functionality is *f3*, since there is not any choice for modifying *f1* and *f2* provider.

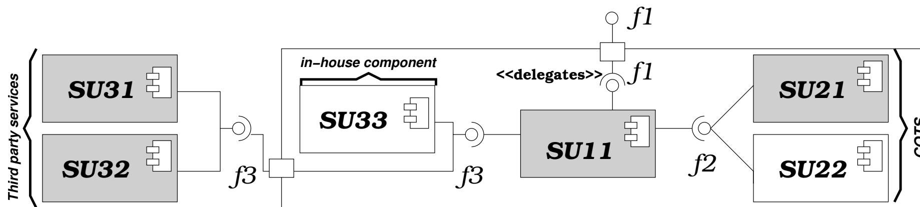


Figure 7.1: System example

In the component diagram we identify a functionality with a service. The interface for an offered service is represented by a ball, while the required one by a semicircle, see Figure 7.2(a). When we need to combine several interfaces for the same service (as in Figure 7.2(b)), we simplify as in Figure 7.2(c), which means that both, *SU11* and *SU12*, need *f3*, which is offered either by *SU31* or *SU32* or *SU33*. We do not make assumptions about which software unit is actually invoked and how.

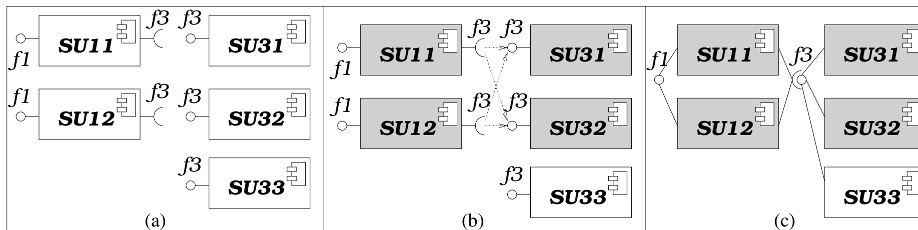


Figure 7.2: How to interpret interfaces

We will also assume: a) a system requiring n different functionalities, $f_i | i = \{1..n\}$; b) the existence of n sets of software units, SU_i, each set offering an f_i. Then, for each f_i, an architect can select a subset of software units of interest, $E_i \subseteq SU_i$ [1].

[1] Note that to create the architecture of a non adaptable functionality f_i, it is enough to select one software unit that offers it; therefore $|E_i| = 1$.

For the sake of simplicity, we avoid to represent software entities devoted to manage the infrastructure of the self-adaptive system[2]. In fact, we consider them as aggregated to the functional software units, i.e., a new software unit is assumed to add to the infrastructure a new proportional complexity for its managing.

It is well-known that there exist software units that expose mechanisms to manage their adaptability. For example, a software unit devoted to perform heavy graphics computations can offer on-demand settings to set the accuracy of the computation. In this case, we assume each on-demand setting as a different software unit providing the target functionality. Naturally, we suppose a discrete and finite number of settings. Figure 7.3 depicts an example for an hypothetical software unit that exposes two settings, i.e., two choices for the system to adapt *f1*.

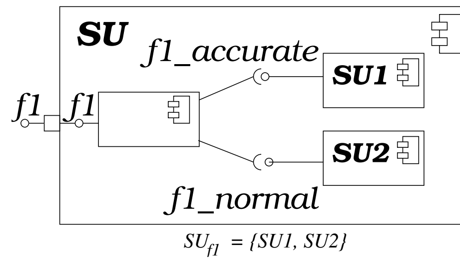


Figure 7.3: Adaptable software unit

7.2.2 Adaptability metrics

Absolute functionality adaptability index (AFAI) represents the number of software units of interest for a given funtionality.

$$AFAI \in \mathbb{N}^n \ | \ AFAI_i = |E_i|$$

Inspired by the *element adaptability index* in [SC01], here a natural number, instead of a boolean one (0 no adaptable, 1 adaptable), quantifies how much adaptable the functionality is.

Referring to the example in Figure 7.1, we observe that $AFAI = [1, 1, 2]$.

Relative functionality adaptability index (RFAI) represents, for a given functionality, the number of software units of interest w.r.t. the number of units actually offering such functionality.

$$RFAI \in \mathbb{Q}^n \ | \ RFAI_i = \frac{|E_i|}{|SU_i|}$$

[2] Those necessary to: make requests compliant with the actual interfaces; monitor the behavior of the functional software units, and; develop the logic that manages the adaptation.

It describes how each functionality stresses its adaptability choices and it informs how much more adaptable the functionality could be. RFAI vector values near to one mean that the system is using almost all the adaptability the market can offer.

Referring to the example we can observe that $RFAI = [1, 0.5, 0.\dot{6}]$.

Absolute software units index (ASUI) represents the number of software units of interest for the system.

$$ASUI \in \mathbb{N} \mid ASUI = \sum_{i=1}^{n} AFAI_i$$

This index offers an overall view about the size of the system, and an insight into the effort the designer needs to invest to manage the whole architecture.

Referring to the example in Figure 7.1, $ASUI = 1 + 1 + 2 = 4$.

Mean functionality adaptability index (MFAI) represents the mean number of software units of interest per functionality.

$$MFAI \in \mathbb{Q} \mid MFAI = \frac{ASUI}{n}$$

This metric offers insights into the mean size and effort needed to manage each functionality.

Referring to the example in Figure 7.1, $MFAI = \frac{4}{3} = 1.\dot{3}$.

Mean of relative functionality adaptability index (MRFAI) represents the mean of $RFAI$.

$$MRFAI \in \mathbb{Q}\{0..1\} \mid MRFAI = \frac{\Sigma_{i=1}^{n} RFAI_i}{n}$$

This index informs architecture managers about the mean degree of utilization of the potential units for each functionality. A value close to one means that the current architecture uses almost all software units in the market. A value close to zero means that: a) the system can be much more adaptable (adding units not yet in the design), b) very different architecture alternatives with the same adaptability metric values can be created.

Referring to the example, $MRFAI = \frac{1+0.5+0.\dot{6}}{3} = 0.7\dot{2}$.

Absolute system adaptability index (ASAI) represents the number of software units of interest w.r.t. the number of available units to architect the system.

$$ASAI \in \mathbb{Q} \mid ASAI = \frac{ASUI}{\Sigma_{i=1}^{n} |SU_i|}$$

Referring to the example in Figure 7.1, $ASAI = \frac{4}{1+2+3} = 0.\dot{6}$.

For ASAI, a value close to one means that the market offers few choices to increase the system adaptability. When a new software unit is bounded to the architecture, ASAI increases in a constant value $(1/\Sigma_{i=1}^{n} |SU_i|)$ regardless of the number of units already considered for the same functionality. Compared to MRFAI, ASAI devises a global view of the system size w.r.t. its maximum reachable size, but does not foretell the amount of different architectural

alternatives the system could reach. To clarify the difference, consider an architecture which includes all available software units for all functionalities but one, however such functionality is realized by a large number of software units while only a few are considered. In such case, ASAI is not close to one (a large number of units are not used), however MRFAI is close to one (all required functionalities, but one, are architected considering their maximum adaptability).

Name	**Range**	**Value**	**Example Fig 7.1**
AFAI	$\mathbb{N}^n$	$\{\lvert E_i \rvert\}$	$[1, 1, 2]$
MFAI	$\mathbb{Q}_+$	$\frac{ASUI}{n}$	$1.\dot{3}$
RFAI	$\mathbb{Q}^n \in \{0..1\}$	$\{\frac{\lvert E_i \rvert}{\lvert SU_i \rvert}\}$	$[1, 0.5, 0.\dot{6}]$
MRFAI	$\mathbb{Q} \in \{0..1\}$	$\frac{\Sigma_{i=1}^{n} RFAI_i}{n}$	$0.7\dot{2}$
ASUI	$\mathbb{N}$	$\Sigma_{i=1}^{n} AFAI_i$	4
ASAI	$\mathbb{Q} \in \{0..1\}$	$\frac{ASUI}{\Sigma_{i=1}^{n} \lvert SU_i \rvert}$	$0.\dot{6}$

Table 7.1: Summary of the metrics

Table 7.1 brings together the six proposed metrics together with the metric values for the example system in Figure 7.1.

7.3 Relating adaptability to a system quality

Bass et al. [BCK05] defend that within complex systems, quality attributes can never be achieved in isolation, the achievement of anyone will have an effect, sometimes positive and sometimes negative, on the achievement of others. Adaptability is not an exception, it can influence other qualities such as performance, reliability or maintainability. An increment in the adaptability can cause an improvement in some of them, but also a damage. So, having measured the adaptability of a system, using some of the metrics in Section 7.2, we want to investigate ways to relate these values to measured values for other qualities. This is important since this relation can assess trade-offs among adaptability and the other qualities. Moreover, from this relation we can study how to obtain thresholds of adaptability for the system w.r.t. quality requirements.

For the sake of simplicity let us start focussing the discussion on a given requirement for a given system. A table to classify requirements can make our proposal easier to be presented (see Table 7.2). A first dimension of the table separates software qualities, so, for a given system:

- some qualities *increase* their measured values when the adaptability increases their owns.
- some qualities *decrease* their measured values when the adaptability increases their owns.

- some qualities do not depend on adaptability variations. We are not interested in this group since we are focussed on the influence of adaptability on the requirement.

A second dimension in the table can consider how the requirement is formulated:

- as *higher than*; e.g., "system availability shall be *higher than* ..."
- as *lower than*; e.g., "system mean response time shall be *lower than*..."

When adaptability increases	Requirement formulated as *Higher than*	*Lower than*
the quality value increases	**Helps**	**Hurts**
the quality value decreases	**Hurts**	**Helps**
the quality is not affected	No effect	

Table 7.2: Effect of adaptability on a quality requirement

Each region of interest in Table 7.2 has been labelled as *Helps* or *Hurts* to indicate the effect of the adaptability on the quality requirement. So we read Table 7.2 as "When adaptability increases, if the quality increases, it *helps* the fulfillment of the requirements of this quality formulated as *higher than*". We do not intend to support the idea that a certain quality always behaves the same. On the contrary, this can be only assessed after analysis, when the evolution of the measures of the quality regarding the values of the adaptability is known. For example, given a requirement, say "response time shall be lower than 3 sec.", we first study in the target system whether the response time increases when the selected adaptability metric increases. In such case, the requirement belongs to the first row, second column, since "when adaptability increases, the response time increases and it *hurts* the fulfillment of the requirement which has been formulated as *lower than*".

However, for another system, it may happen that this requirement can be *helped* by increments in the adaptability. Even more, for the same system, a requirement could be in *Helps* or *Hurts* depending on the system operational profile. For example, consider a system that balances its workload. For high workload, the response time will decrease when the system adapts and balances its load; then adaptability *Helps* response time. Nevertheless, for low workloads the response time will remain about the same whether the system balances the load or not, but balancing operations will add execution overhead; so the execution time will be higher and response time can belong to *Hurts*. From these examples, we conclude that the effect of adaptability upon a requirement cannot be allocated in a concrete region before to carry out an analysis.

If a linear relation between adaptability and a quality existed, then we could get a graph like the one in Figures 7.4(a) and (b) (for the first and the second rows in Table 7.2, respectively). However, the software units and their connections may have a more profound effect into the quality of the system than the adaptability values. The extreme case for this affirmation is depicted in Figure 7.4(c), where software units have all effect into the quality and the

amount of system adaptability has not effect at all on the system quality. However, we can find some promising adaptability intervals for the cases when the adaptability and the quality are not completely independent. Figures 7.5 and 7.6 depict these situations.

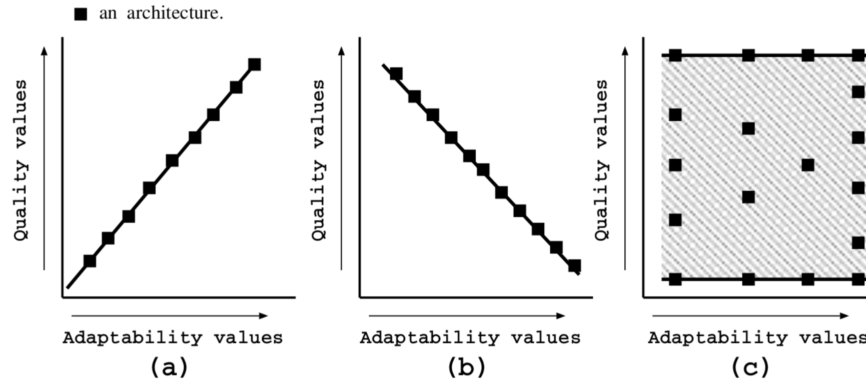
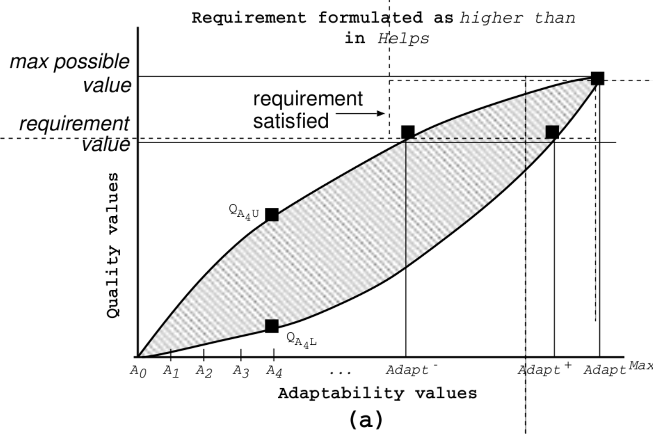


Figure 7.4: Adaptability and quality relationships

Interpretation of the graphs Figure 7.5(a) depicts a generic graph for quality requirements that belong to the first *Helps* (so, those which the quality to which they belong increases when the adaptability increases and are formulated as *higher than*). X-axis are increasing values, A_i, of an adaptability metric formulated in Section 7.2. Y-axis represents values for the target quality. $Adapt^{Max}$ is the maximum adaptability value the architecture manager considers. When it is not explicitly defined, then we will consider all architectural combinations (e.g., in case of ASAI, MRFAI or ASUI metrics such value would be 1, 1, and $\Sigma_{i=1}^{n} |SU_i|$, respectively). For each A_i, we are interested in two values: the upper bound, Q_{A_iU}, (maximum value of the quality that an architecture with adaptability A_i could reach) and the lower bound, Q_{A_iL} (minimum value). In between these two values there exist a number of architectures that exhibit the same adaptability but different quality values.

In graph (a), the requirement to fulfill is called *Requirement value*. $Adapt^{-}$ is the lowest A_i that can satisfy the requirement, and $Adapt^{+}$ represents the lowest A_i whose lower bound also satisfies the requirement, which indeed is satisfied until $Adapt^{Max}$. These values mean that to fulfill the requirement, the system must have at least adaptability $Adapt^{-}$, and, any selected architecture with at least $Adapt^{+}$ will also satisfy it. For adaptabilities between these values, there will be architectural alternatives that will satisfy the requirement and others that will not.

Figure 7.5(b) depicts the graph for requirements belonging to *Hurts* whose quality increases with adaptability and are formulated as *lower than*; for example, given as requirement *system cost should be lower than...* and happening that the more adaptable the system is the more expensive it becomes. Then, in this case, any architecture with adaptability $A_i \leq Adapt^{-}$ fulfills the requirement. $Adapt^{+}$ is the maximum A_i for which we know



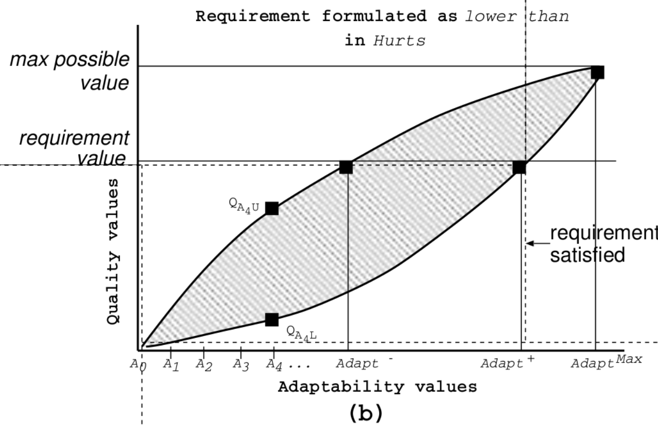


Figure 7.5: Relation among adaptability and other quality

that exists some architecture that satisfies the requirement. The graphs for the remaining two cases appear in Figure 7.6.

These four cases could be merged into only two. If we negate the qualities in the second row in Table 7.2, then we get the first row, so the graphs in Figure 7.6 could be substituted for those in Figure 7.5. However, it is difficult to defend that any quality has this counterpart. For this reason, we prefer to consider all the four cases.

Method to create a graph The assessment of a given requirement (or trade-off among adaptability and a property) implies to generate the graph for the system under study. We start with $A_i = A_0$, and then compute Q_{A_0U}, Q_{A_0L}. Next, the adaptability value is in-

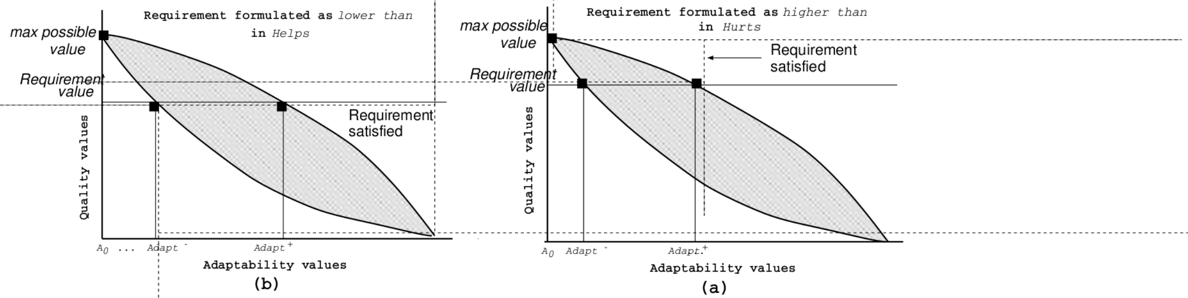


Figure 7.6: Relation among adaptability and other quality: rest of the cases

creased and the next smallest value $A_i = A_1$ and its Q_{A_1U} and Q_{A_1L} are calculated. The process continues until A_i reaches $Adapt^{Max}$. These results generate the actual upper and lower bound curves. Depending on whether the target quality belongs to *Helps* or *Hurts*, we consider as $Adapt^-$ either the lowest A_i that can satisfy the requirement, or the highest A_i where both Q_{A_iU} and Q_{A_iL} satisfy the requirement. Alike, to calculate $Adapt^+$, either we consider -in case the quality belongs to *Helps*- the lowest A_i where both Q_{A_iU} and Q_{A_iL} bounds satisfy the requirement, or -in case of belonging to *Hurts*- the highest A_i that can satisfy the requirement.

7.4 Example

This section presents an example to study trade-offs between the adaptability of a system and the availability and cost it presents. The considered example studies a web system used by students to register with the University for an academic year. The system is composed of: a presentation and notification layer with the web interface and the mechanisms to communicate with the student, and the application logic layer with the rules that approve or reject the students' proposal list of subjects to take. At first, the students interact with the first layer and give their info and proposal of subjects for their degree course. This layer gets the info and passes it to the application logic. If the University rules are not incompatible with the student's choice, this layer interacts with the bank web application to proceed with the course payment. Once it has finished, the application control comes back to the presentation and notification layer. This layer finally sends a message and an email to the student with information about their registration process. For this example, we assume that there exist two components that implement exclusively the application logic; a component that implements exclusively the presentation and notification layer; a component that implements both the presentation and notification and the application logic layers; two services that implement the bank payment (two banks for paying) and three software applications that offer the email sending, one of them is the local to the University and two of them are third-party email providers.

Figure 7.7, depicts the component diagram of the described system using the generic notation with SU. Relation of generic notation to web application example is described in

Table 7.3.

Web application example	Generic Notation
Student registration	f_1
Student requirement satisfaction	f_2
Send email	f_3
Bank payment	f_4
Presentation and notification component	*SU11*
Presentation and notification + application logic component	*SU12*
Application logic 1	*SU21*
Application logic 2	*SU22*
Third-party email provider 1	*SU31*
Third-party email provider 2	*SU32*
Local email provider	*SU33*
Bank 1 payment service	*SU41*
Bank 2 payment service	*SU42*

Table 7.3: Notations relationship

Note that Figure 7.7 slightly increments Figure 7.1 example by adding functionality f_4 and software units $SU12$, $SU41$ and $SU42$. We assume that the quality requirements to fulfill are *the system availability shall be higher than 0.9*, and *the system cost shall be lower than 30 monetary units*. Availability means "readiness for correct service" [ALRL04], while the cost property simply describes the price of the set of software units that will make up the system. For the rest of the example, we differentiate software units as *terminals* and *non-terminals*. Terminals are those not needing other functionalities (e.g., *SU31* or *SU32*), while non-terminals do need (e.g., *SU11* or *SU12*). Note that in the example some software units offering the same functionality are not completely *replaceable*. For example, *SU11* cannot completely replace *SU12*, since the former needs f_2 but not f_4, and the later needs the opposite.

System operational profile In the components diagram we depict the quantitative information needed to compute the system availability and cost. For simplicity, this information appears inside the components and in the contiguous table. However a more formal approach, like the MARTE [Obj05] standard profile, could be used.

- $P_i^{f_j}$ means for a software unit i, the probability of requiring functionality j.
- $N_i^{f_j}$ means for a software unit i, the mean number of requests to functionality j.

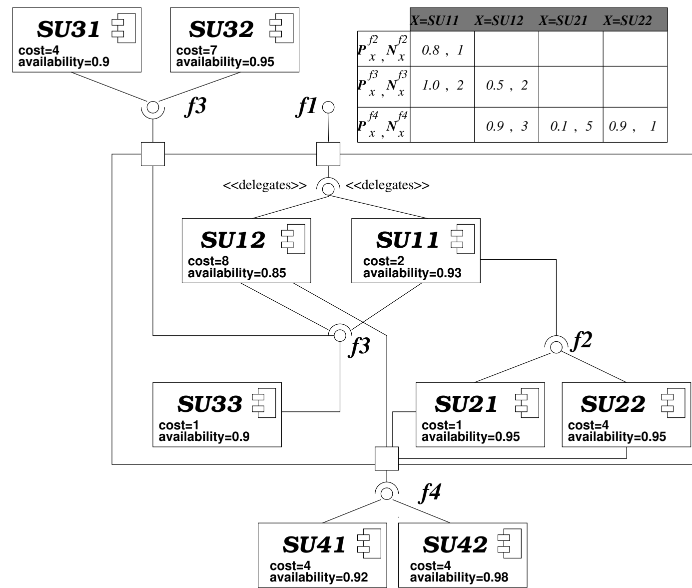


Figure 7.7: The architecture of a complex -w.r.t. adaptability- system

- The *availability* of a software unit is a measure obtained from the third-party provider or monitoring the software unit.
- The *cost* of a software unit is the value we pay to a third-party provider for using it.

$P_i^{f_j}$ and $N_i^{f_j}$ could be combined to form the "mean number of requests per execution", however we prefer to keep them separated for the sake of system availability computation. For example, we could need to call a software unit only for the 20% of our executions, but we have to call it five times per execution; for another software unit, it may happen that we have to invoke it once for each execution. In both cases the "mean number of requests" is one. However, in the latter case all the system executions are prone to fail depending on the availability of the software unit, while in the former, the remaining 80% of executions are safe.

7.4.1 Computation of system qualities

Availability computation.

Definition 7.1. *The availability of a* terminal *software unit is the one annotated in the diagram. E.g.,* $Av(SU31) = SU31.availability = 0.9$.

Definition 7.2. *The availability of a* functionality f_i *is:*

$$Av(f_i) \ = \ 1 - \prod_{su_{ij} \in E_i} (1 - Av(su_{ij})) \tag{7.1}$$

A functionality is available if any of the components in E_i (as defined in Section 7.2) is available.

For example, if $E_3 = \{SU31, SU32\}$, then $Av(f_3) = 1-(1-0.9)\cdot(1-0.95) = 0.995$ (note that all components providing $f3$ are terminals).

When E_i includes non-terminals, we previously compute the availability of the non-terminals.

Definition 7.3. *The availability of a* non-terminal *is:*

$$Av(su_{ij}) = su_{ij}.availability \ \cdot \prod_{f_k \in RF(su_{ij})} ((1 - P_{ij}^{f_k}) + P_{ij}^{f_k} \cdot Av(f_k)^{N_{ij}^{f_k}})$$

where $RF(su_{ij})$ is the set of functionalities su_{ij} requires.

For example, $RF(SU12) = \{f_3, f_4\}$, if we consider $E_3 = \{SU31, SU32\}$ and $E_4 = \{SU42\}$. Then, $Av(f_4) = 0.98$ and $Av(SU12) = 0.85 \cdot ((0.1 + 0.9 \cdot 0.98^3) \cdot (0.5 + 0.5 \cdot 0.995^2)) = 0.801$.

Definition 7.4. *The system availability is recursively computed from the main functionality using equation (7.1).*

In our example, if we suppose the architecture made of *SU12*, *SU31*, *SU32* and *SU41*, then the result is: $Av(f_1) = 1 - (1 - Av(SU12)) = 1 - (1 - 0.801) = 0.801$.

Cost computation.

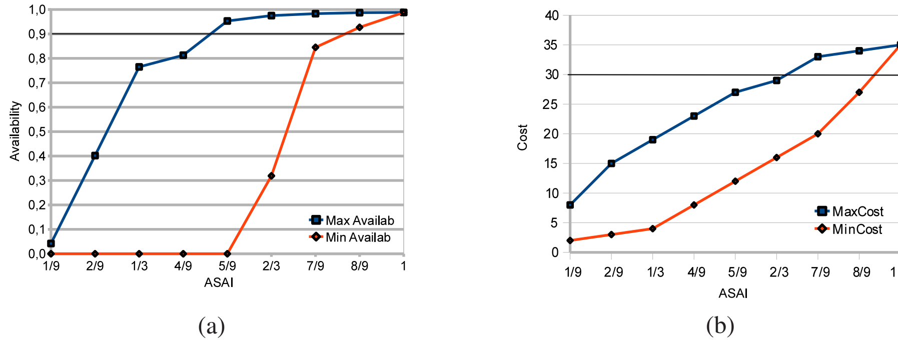
Definition 7.5. *The cost of the system is:*

$$Cost = \sum_i \sum_{c_{ij} \in E_i} c_{ij}.cost$$

For example, the cost of a system made of *SU11*, *SU21*, *SU31*, *SU32* is $2+1+4+7 = 14$ monetary units.

The calculation of the availability is a simple but interesting method we propose in this example. However, for the calculation of the cost, we recognize it to be simplistic[3], yet we consider that the focus of the work is on trade-offs between qualities.

[3] We have not considered deployment costs, developed cost or distinguished among advanced payment manners to service providers such as payment for execution requests, payment for temporal contract or payment for a COTS component acquisition.



(a) (b)

Figure 7.8: (a) Relation among ASAI and availability (b) Relation among ASAI and cost

7.4.2 Relation of adaptability to availability and cost

Applying to our example the models of computation in Section 7.4.1 we discovered that the availability requirement belongs to *Helps* since the availability increases when the adaptability does and it is required a value *higher than* a given threshold (0.9). On the other hand, the cost requirement belongs to *Hurts*, since the cost increases when the adaptability does and the requirement is *lower than*.

From the metrics presented in Section 7.2 we have used ASAI (the number of software units that compose the system w.r.t. the number of the software units that could be used). Following the method presented in Section 7.3 we created the corresponding graphs, we started considering the architecture made of only one unit $A_0 = 1$. Moreover, since we are not assuming a maximum adaptability, then $Adapt^{Max} = 1$.

It is worth noticing that although any metric in Section 7.2 can be used for this study, we have chosen one of the simplest for the sake of clarity. For example, vectorial metrics prevented us from showing the results in graphical form, since there would be necessary $n + 1$ dimensions to depict the relation, while scalar ones are represented in 2D graphs.

Graph to relate adaptability and availability

For $A_0 = 1$, the selected software unit should be the one providing the main functionality f_1, i.e., *SU11* or *SU12*. *SU11* shows an availability equals to 0, since it needs to request f_3 with probability 1, but f_3 is not available at present. *SU12*, instead, shows an availability equals to 0.0425.

Figure 7.8(a) depicts the lower and upper bounds of the system availability for each value of ASAI. The graph shows the existence of solutions satisfying the requirement, i.e., architectures with availability higher than 0.9. The first suitable solution is for an ASAI equals to $\frac{5}{9}$ software units, then $Adapt^{-} = \frac{5}{9}$. In this case the availability is 0.954 and the architecture is made of $E_1 = \{SU11, SU12\}$, $E_2 = \{SU22\}$, $E_3 = \{SU32\}$ and $E_4 = \{SU42\}$. For informative purposes we computed all the metrics for this solution, see second column in Table 7.4.

Regarding $Adapt^+$, the graph clearly shows that all architectures with ASAI $> \frac{7}{9}$ fulfill the required availability[4]. For ASAI $= \frac{8}{9}$, the lower bound (worst architectural alternative) offers an availability of 0.9271. In such case, the system is made of $E_1 = \{SU11\}$, $E_2 = \{SU21, SU22\}$, $E_3 = \{SU31, SU32, SU33\}$, and $E_4 = \{SU41, SU42\}$. Third column in Table 7.4 shows the other metric values for this architecture.

Graph to relate adaptability and cost

We apply again the method in Section 7.3 to compute in this case the cost of the system for each value of ASAI, from $A_0 = \frac{1}{9}$ to $Adapt^{Max} = 1$. Results in Figure 7.8(b) show that it is possible to find solutions satisfying the requirement up to an ASAI $= \frac{8}{9}$. Moreover, all architectures with ASAI lower than $\frac{7}{9}$ will satisfy the requirement. Again, for informative purposes, we computed all the metrics for the values of $Adapt^-$ and $Adapt^+$, they appear in Table 7.4.

$$n = 4 \qquad |SU_1| = 2 \qquad |SU_2| = 2 \qquad |SU_3| = 3 \qquad |SU_4| = 2$$

	Availability		**Cost**	
	$Adapt^-$	$Adapt^+$	$Adapt^-$	$Adapt^+$
$AFAI$	$[2,1,1,1]$	$[1,2,3,2]$	$[2,1,2,1]$	$[1,2,3,2]$
$RFAI$	$[1,\frac{1}{2},\frac{1}{3},\frac{1}{2}]$	$[\frac{1}{2},1,1,1]$	$[1,\frac{1}{2},\frac{2}{3},\frac{1}{2}]$	$[\frac{1}{2},1,1,1]$
$ASUI$	5	8	6	8
$MFAI$	1.25	2	1.5	2
$MRSAI$	$0.58\dot{3}$	0.875	$0.\dot{6}$	0.875
$ASAI$	$0.\dot{5}$	$0.\dot{8}$	$0.\dot{6}$	$0.\dot{8}$

Table 7.4: Metric values of the architectures in Figure 7.8 for bounding values $Q_{Adapt-U}$ and $Q_{Adapt+L}$

Adaptability, availability and cost

Putting together both studies, we can foretell that:

- No suitable architecture can be found for an ASAI $< \frac{5}{9}$ or an ASAI $= 1$, since either the requirement of availability or the one of cost cannot be satisfied.
- There are suitable architectures for values of ASAI $= \frac{5}{9}$, ASAI $= \frac{6}{9}$ and ASAI $= \frac{8}{9}$.
- There can exist suitable architectures for ASAI $= \frac{7}{9}$.

[4] We remark that, following indications in Section 7.3, the non-suitable architectural alternatives have been discarded.

7.5 Relating quality requirements

We have hitherto proposed an approach to relate system adaptability to availability and cost. To make this approach easily applicable our goal is to define an automated framework that effectively assists to architect a system that meets several quality requirements within an adaptability threshold. To this end we formalize below these relationships.

Definition 7.6. *Let us define $Reqs$ as the set of the requirements of the system. R_{Helps} and R_{Hurts} as the requirements that respectively belong to $Helps$ and $Hurts$ as in Table 7.2. Then, $Reqs = R_{Helps} \cup R_{Hurts}$.*

Definition 7.7. *$\forall req \in R_{Helps}$, we define $MAdapt^- = max(Adapt^-)$ and $MAdapt^+ = max(Adapt^+)$ and $\forall req \in R_{Hurts}$, we define $mAdapt^- = min(Adapt^-)$ and $mAdapt^+ = min(Adapt^+)$.*

Definition 7.8. *We define $ADAPT(arch)$ as the adaptability value of architecture $arch$, while $SAT(arch, req)$ means that $arch$ satisfies req.*

Proposition 7.9. *When*

$$(MAdapt^- \leq mAdapt^-) \vee (MAdapt^+ \leq mAdapt^+) \tag{7.2}$$

then $\forall A_i \in [MAdapt^-, mAdapt^-] \vee [MAdapt^+, mAdapt^+],\ \exists arch \mid ADAPT(arch) = A_i \wedge \forall req \in Reqs, SAT(arch, req)$.

Proposition 7.10. *When*

$$(MAdapt^- \leq mAdapt^-) \wedge (MAdapt^+ \leq mAdapt^+) \tag{7.3}$$

then $(\forall A_i \in [MAdapt^-, mAdapt^-] \cap [MAdapt^+, mAdapt^+]) \wedge (\forall arch \mid ADAPT(arch) = A_i) \longrightarrow \forall req \in Reqs, SAT(arch, req)$.

Obviously, (7.2) and (7.3) are *sufficient conditions*, when they do not hold, we can at least assess whether it is impossible to fulfill the requirements. This is guaranteed by proposition 7.11.

Proposition 7.11. *When*

$$mAdapt^+ < MAdapt^- \tag{7.4}$$

then $\nexists arch \mid \forall req \in Reqs, SAT(arch, req)$.

Otherwise, if neither (7.2) nor (7.4) hold, then it cannot be proved the existence or absence of architectures that satisfy the requirements. However, if such solutions exist, then their adaptability values must belong to the interval:

$$[MAdapt^-, mAdapt^+] \cap [mAdapt^-, MAdapt^+] \tag{7.5}$$

Proof. Section 7.5.1 sketches the demonstrations of 7.9, 7.10 and 7.11 graphically. □

7.5.1 Graphical representation

We use the location in the graph of $MAdapt^{-}$, $MAdapt^{+}$, $mAdapt^{-}$ and $mAdapt^{+}$ to show the propositions above. So, they can be arranged up to $4!$ different permutations. However, since by definition $MAdapt^{-} \leq MAdapt^{+}$ and $mAdapt^{-} \leq mAdapt^{+}$, then the amount of permutations is reduced to $\frac{4!}{2!2!} = 6$. Figure 7.9 depicts these six possible scenarios. Suitable architectures appear only in scenarios (a),(b),(c) and (e). For the sake of simplicity, we have considered $|R_{Helps}| = |R_{Hurts}| = 1$. The symbols in the figure have to be interpreted as follows:

- The symbol '$\exists$' represents a region where condition (7.2) holds, then ensuring the existence of at least one architectural solution for each adaptability value in such interval.
- The symbol '$\forall$' represents a region where condition (7.3) holds, then ensuring that all architectures within the region satisfy all requirements.
- The symbol '$\nexists$' represents a region where none architecture satisfies all requirements. For example, in Figure 7.9(f) the $\nexists$ symbol covers the entire region because here formula (7.4) holds.
- The 🔎 symbol represents a region where condition (7.5) holds, then it is not possible to prove the existence or absence of architectures satisfying the requirements.

7.6 Conclusion

In this chapter, we have presented a set of metrics helpful to quantify and evaluate the adaptability of software systems at the architectural level. Besides, we have defined a relationship between these metrics and the quality requirements of the system. The approach can be used during design time to help software architects in the generation of a suitable adaptable architecture.

One of the aims related to the definition of metrics is the possibility to have some means for the evaluation and comparison of different systems in terms of adaptability and quality requirements. A trade-off analysis in this case should be carried out in order to take the decision that better fulfills the various stakeholders needs.

The approach can be improved along several directions. We are extending the set of metrics and applying them to a great many case studies to guarantee their usefulness also from a statistical viewpoint. Specifically, we are working towards the inclusion of aspects such as the "criticality" or "importance" of the offered functionalities, adding for example, some reasonable weights to our metrics and defining new metrics that include this concept. We plan also to relax the constraint requiring that each software unit offers a single functionality. This is not a trivial step since it requires to take into account also the interdependencies between the different offered functionalities and their quality requirements and it would probably entail a definition of new adaptability metrics.

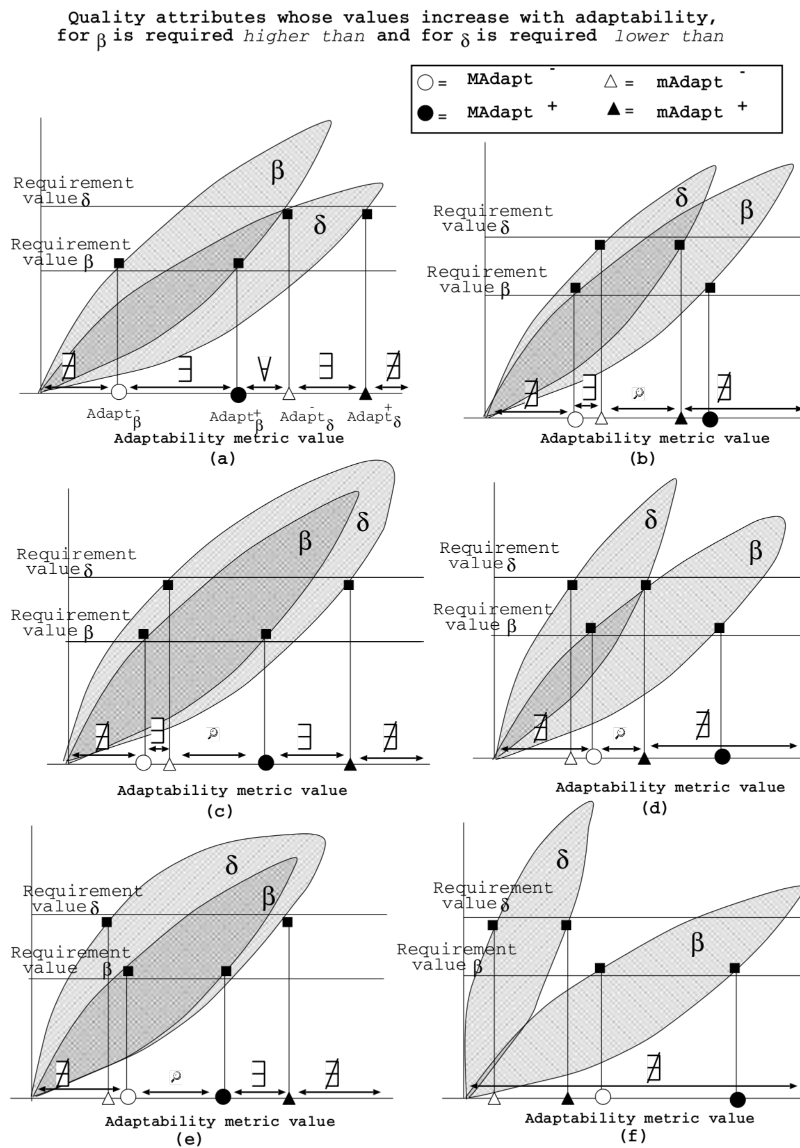


Figure 7.9: Generic graphs showing relations between adaptability and other quality requirements

Another direction that deserves further investigation is the integration and combination of our metrics with the ones proposed in other works (e.g., [KRG+10]) then empowering soft-

ware architects to compare adaptive system designs with the system design without adaptability.

Finally, the development of an automatic tool that implements the approach would bring several benefits. One of these benefits is that this tool would allow the integration of the approach at runtime -when human intervention is not possible-. Autonomic systems would benefit from the integration at runtime since they can need to re-architect themselves, and this approach would offer a set of suitable adaptability ranges to guide the re-architecting process.

7.7 Related work

The set of metrics we offer in this chapter is strongly inspired by the one in [SC01], work in which authors also give a set of metrics for adaptability applicable at architectural level. Our extension aims at supporting a higher degree of quantification starting from the most basic metric. In our approach, the metric itself does not only track if a required functionality of the system is adaptable or not, but we also quantify *how much* adaptable it is by means of a natural number. The same authors propose in [CS01] a framework that is a specialization of a general qualitative framework to reason about non-functional requirements [CNYM99, MCL+01]. That framework is concentrated on adaptability requirement and works with quantitative values. Our work, on the contrary, is based on the addition of adaptability property to systems in order to make such systems able to meet also the non-functional quality requirements.

In [RM09, KRG+10], the authors wonder whether it is possible to measure and evaluate the adaptability of systems in order to compare different adaptive solutions. To take a step forward, they propose a set of quantitative metrics grouped by categories. These metrics are calculated statically. However, their approach can be extended to be applicable in a dynamic environment. In this direction, we foresee a possible integration between the metrics defined in this chapter and the approach proposed in [RM09, KRG+10]. Indeed, our approach can be used to discover which are the suitable architecture adaptabilities that can make the system able to meet the desired quality non-functional requirements, and then use their higher-level metrics to offer an evaluation and comparison of the already calculated suitable architectures.

In [RWvM10], authors define a methodology to evaluate the adaptivity of a system. This evaluation is based on measurement traces or simulation traces that can be obtained, in test-beds, real systems or software tools for discrete-event simulation. Besides, this methodology is enhanced with the definition of a simple metric that evaluates adaptivity on a scale from zero to one.

In [YHZ+09] a trade-off analysis among quality properties of adaptive systems is presented. This approach takes into account changes in runtime contexts and the decision to adopt an adaptation strategy is performed during runtime, when the system knows the current real context. To achieve that goal, the authors propose a three-phases methodology, where the firsts two phases are done during design time and the third one is executed by the system during runtime. The phases are: (i) analyze the target architecture to find trade-off points among qualities, (ii) design different adaptive strategies and record them in the architecture model, (iii) deploy the system collecting context information and selecting the best strategy during runtime.

With respect to existing work, in this chapter we propose a more extensive set of architectural metrics that can be used in the quantitative evaluation of software adaptability. These metrics have been empowered with the definition of explicit relationships between adaptability and quality values, such as availability and performance.

Part II

Model-Driven Engineering for QoS Evaluation

This part of the thesis explains our research in the Model Driven Engineering (MDE) [Sch06a] field and in the Model Driven Development (MDD) [AK03] techniques aimed at software QoS evaluation. These techniques usually focus on a transformation path from high level models to platform specific models (down to the executable code) of a software system.

Model-based evaluation of extra functional properties -such as performance, reliability or energy consumption- usually needs analyzable models, but software developers may not hold expertise in creating and analyzing these models. To overcome such limitation, it has emerged the idea of exploiting MDD for QoS assessment. This idea conceives a special type of transformation path whose source and target are a software design model and an analyzable model, respectively. For example, the design model can be a set of UML diagrams and the analyzable model can belong to queueing networks, Markov chains, Petri nets or process algebras families. Since these transformations are automatic, software developers are released from the manual creation of analyzable models.

Recently, some challenges that hamper the implementation of model transformations have been realized. Two of these challenges are:

- A software design has much more information than the analyzable model requires, which makes transformations intricate. Besides, the relevant information may be spread along many software design elements. For example, in a UML software design, performance relevant information can be scattered throughout activity, sequence, component, state, deployment and use case diagrams. The transformation process has to find the performance relevant information in different models and create the performance analyzable model. So, even if the core theory of the transformation was not hard to follow, these transformations would be hindered by the amount of useless information in the design model for QoS evaluation.

- There are many software design languages and, according to [GMS07a], even for the same project, a different language can be used in different stages of software development. Besides, since a particular property may be better analyzed by using a particular language, there should be implemented transformations from each design language to several analyzable models. These circumstances force the need of a transformation path from any design language to any analyzable language. Therefore, they are required *M-by-N* transformations, being M the amount of design languages and N the amount of analyzable languages.

To face these challenges, there have been proposed intermediate models that are placed between design and analyzable models, then splitting the transformation process up into two,

presumably easier, steps. The first step implements a transformation that gathers the information relevant for QoS evaluation into a single model whilst it abstracts away from the QoS irrelevant design concepts. The second step implements a transformation between the intermediate model to the analyzable model.

This procedure helps to face the previous challenges because: the intermediate model is free of irrelevant information for QoS evaluation; a single model keeps all the required information; and sharing a pivot language, the amount of required translations is reduced to $M + N$.

Since most of the research in the previous part of this thesis relies on the analysis of software models with formal methods, a number of model transformations of this type had to be done. We have contributed to this research field by describing two model-to-model transformations. For example, this work is helpful for the task of the uppermost layer of the reference architecture in Chapter 2; see that the *Adaptation Plan Generator* uses software models for evaluating the performance of a software workflow or the expected energy consumption. These software models are converted into a model in an analyzable language, which in our case was stochastic Petri nets, and therefore results of extra functional properties could be obtained.

The first model transformation we propose uses the intermediate model called Core Scenario Model (CSM)[PW07]. CSM is a scenario oriented model that represents software behavior together with performance and resource usage information. So, this language is specially useful to evaluate performance of execution scenarios of software. Our research in Chapter 8 proposes a transformation that takes a source model in CSM language and transforms it into an analyzable GSPN. We also present and describe a software tool that implements the transformation theory. Then, using the transformation from UML diagrams annotated with performance information into a CSM proposed in [PW07] and our tool, we can complete the transformation path from software design models to performance analyzable models. This is useful, for example, for the Petri nets to analyze in Chapter 3.

The second model transformation uses KLAPER [GMS07a] as pivot language, which stands for Kernel LAnguage for PErformance and Reliability analysis. KLAPER creates resource oriented models that represent the relevant information for the analysis of software extra functional properties. This resource oriented representation of software makes KLAPER be recommended for designs that follow component-based software engineering methodologies. Concretely, this thesis uses the dynamic version of KLAPER, called D-KLAPER, which allows modeling dynamic bindings between resources. Then, D-KLAPER models allow us to represent characteristics related to self-adaptations of software. In Chapter 9, we extend the modeling power of D-KLAPER to allow it to model reactive software. After, we use the proposed D-KLAPER extension as intermediate language for the transformation path between a design model of reactive software (represented as UML state machine diagrams) and GSPNs. We accordingly describe our research in two model-to-model transformations: a transformation from the most typical characteristics of UML state machine diagrams to the extended D-KLAPER, and from the extended D-KLAPER to GSPNs. Finally, the translation theory is applied to an example of reactive self-adaptive system whose QoS goals to satisfy are performance and cost.

Related work

Modeling frameworks for dynamically changing software systems have been already proposed. Some of them are mainly targeted to the analysis of functional requirements [ADG98, BCDW04, IFMW08, KM07], and hence are not suitable for the effectiveness analysis of such systems with respect to performance or dependability.

To the best of our knowledge, model-to-model transformation theories and the analysis methods of the generated performance models surrounding CSM and KLAPER languages are two of the most comprehensive approaches in MDD for Software Performance Engineering (SPE) [Smi90]. Furthermore, KLAPER and its KlaperSuite [CFD+11] are not only conceived for performance analysis but also for reliability and other non-functional properties.

The work in [DSP11] presents PCM, another intermediate performance model, and its translation to non-markovian SPNs. The modeling power of the PCM is similar to the CSM, but the CSM details some concepts such as the acquisition and release of resources or the starting and end of the scenario, which implies differences in the translation. PCM uses the UML-SPT profile while we use the more recent MARTE. Although work in [DSP11] claims that the translation to Petri nets has been accomplished, the topic of the paper is not to discuss the design of a tool, however this is one of the main aspects in our work in Chapter 8.

Proposals in [CPR07, HBR+10] value the execution environment at design level for the performance prediction. The implementation of this aspect may improve our tool, however we consider it as future work since theoretical work needs to be invested prior its implementation in the tool.

In [CMI11], it is presented the state of the art in model based analysis techniques with a particular focus on performance issues.

Chapter 8

Model To Model Transformations: From CSM To GSPN

In this chapter we present part of our research in model-to-model transformations for software performance evaluation. We describe the transformation theory between an intermediate model and an analyzable model. The source model is a Core Scenario Model (CSM), which is a scenario-oriented intermediate performance model that filters out the information unrelated to performance in a software design, and the target model are GSPN, which are a formal and analyzable modeling paradigm that have been shown feasible for software performance evaluation. In this chapter we also describe a tool that implements such transformation.

8.1 Problem description

The assessment of software non-functional properties such as performance, is a challenging issue for the software engineering community. Software Performance Engineering (SPE) [Smi90] promotes the use of standard design languages like the Unified Modeling Language (UML) [BJR99] and associated OMG standard profiles, with the aim of leveraging software designs for a prediction of system performance [BDIS04]. The OMG-MARTE (Modeling and Analysis of Real-Time and Embedded systems) [Obj05] profile augments a UML design with information relevant for performance prediction.

In the work carried out in Part I, we needed to evaluate performance of software systems under different configurations and execution environments. Owing to the research already done in the SPE community, we decided to heed their proposals to ease our task.

SPE proposals work on both types of model-driven evaluation: direct transformations of design models into performance models, and transformation paths that include intermediate models between design and performance models. The work in [BDIS04] summarizes some of the main proposals regarding the former approach, while for example, works in [DSP11, GMS07a, WPP+05] follow the latter. As motivated in previous introduction, we decided to follow the latter approach. In this chapter, we present our research in the model-to-model

transformation from the intermediate model CSM [PW07] to the analyzable model GSPN [AMBC+95] and its automation.

CSM language is fully described in [PW07]. A CSM model represents software execution scenarios in terms of well-known performance concepts: steps, workloads, path connectors or resources. In [PW07] it is also explained the model-to-model transformation from UML designs to CSM. Besides, CSM language was proposed in [WPP+05] as intermediate language within the Performance by Unified Model Analysis (PUMA) framework, which is an open architecture that enables the integration of performance analysis in different kinds of software design tools. In that work, there are outlined the model-to-model transformations from CSM to queueing network models (QN), layered queueing network models (LQN) and GSPN.

This chapter extends the description of CSM to GSPN presented in [WPP+05], it presents the tool and algorithms we have developed for the transformation of a CSM into a performance model in terms of GSPN. The GSPN can be analyzed or simulated by using engines such as GreatSPN [Gre] or TimeNET [ZFGH00]. Our tool implements a CSM-GSPN "compositional" translation and uses software standards, such as XML [XML] format. This fact also enables a future integration of the tool within the PUMA architecture. The compositional issue forced us to define a composition operator for GSPN, we present it in appendix B.

The rest of the chapter is organized as follows. Section 8.2 recalls the needed background, the CSM meta-model. Section 8.3 details how CSM concepts are converted into the GSPN models. Section 8.4 describes the tool issues and lessons learned in its development. Section 8.5 applies CSM to GSPN transformation to a case study taken from literature [XWP03], presents performance results and compares them with those obtained using layered queuing networks (LQNs) as performance model. Section 8.6 gives a conclusion.

8.2 Core Scenario Model

The goal of the CSM metamodel is to capture the essentials for building performance models. The class structure of the CSM is shown in Figure 8.1, it corresponds to the one presented in [PW07], however we have added two new abstract classes: the VertexOperation and the ResourceManager.

The CSM represents the performance Scenario flow via a PathConnection type. There is a PathConnection object between each pair of VertexOperations. Indeed, the VertexOperation class has been introduced to distinguish Steps from ResourceManagers. So, a Step is a sequential piece of execution which may in turn be refined as a scenario. While a ResourceManager only manages the resource utilization of steps. The subtypes of PathConnections correspond to the common sequential relationships: branches, merges and forks and joins. A scenario has a Start point and End points. Start points may associate a Workload, then representing the scenario usage. There exist two kind of Resources: Active, which execute steps, and Passive, which are acquired and released during scenarios by special ResAcquire and ResRelease operations. Steps are executed by (software) Components which are passive resources. In turn, Components are associated to the ProcessingResource in which are hosted.

CSM meta-classes own attributes defining their properties. For example a Step has the HostDemand to describe its aggregate use of its host resource (CPU). Table 8.1 gathers the

Figure 8.1: Core Scenario Model

attributes used in this work, the whole list can be found in [PW07].

Figure 8.13(a) depicts a simple example of CSM, where we observe the start and end Vertices, the acquisition and release of a resource, the workload and a couple of Steps -cycleInit and procOneImage-. Figure 8.13(b) is a more complicated CSM which indeed refines the procOneImage Step in Figure 8.13(a).

8.3 CSM meta-classes translation

This section presents the set of patterns that we propose to represent CSM meta-classes. These patterns are at the core of the translation and they will be used by our tool to convert a CSM model into a GSPN [AMBC$^+$95]. A pattern in this context is actually a GSPN subnet parameterized with the attributes of the CSM meta-class the subnet represents.

The places and transitions of the GSPN subnets are named as $name \mid label_1 \mid ... \mid label_n$ (e.g., $t1|getImage|disk_release$). These labels refer names of the CSM model elements and they will be useful to keep track of the CSM operations and to eventually compose the subnets. In the following, we describe for each CSM meta-class, the

CSM meta-class	Attribute name
Scenario	ID[1], Name
GeneralResource	ID, Name, Multiplicity
ActiveResource	*Same as GeneralResource* + OperationTime
ProcessingResource	*Same as ActiveResource*
VertexOperation	ID, Probability
Step	Name, HostDemand, ExtOp, RepetitionCount
ResourceManager	ResourceUnits, Priority
OpenWorkload	ID, ArrivalPattern[2], ArrivalParameter[3]
ClosedWorkload	ID, Population
Message	ID, Kind[4]

1. ID means a unique identifier.
2. [Poisson | periodic | phase-type]
3. The *ArrivalPattern* description.
4. [async | sync | reply]

Table 8.1: Some attributes of some CSM meta-classes

GSPNs subnets patterns implemented by the tool.

8.3.1 Step translation

We revise now all different options to translate a Step. The choice is based on the values of the attributes of the Step: Name, HostDemand, ExternalOperation, RepetitionCount and Probability. The latter inherited from VertexOperation. The ExternalOperation refers to a service external to the model.

- Figure 8.2(a) depicts the subnet for a "dummy" step, i.e., a step that only has a Name but the rest of the attributes are not used. This step, as explained in [PW07], is only used to link PathConnectors, e.g., for linking two consecutive branches. The translation provides a subnet with one place and one transition, both labeled with the name of the Step.
- When the Step declares a Name and a HostDemand, then the subnet is the one in Figure 8.2(b). Transition $t2$ is an immediate one and it will be useful to acquire the processor or host where the Step executes. Transition $t1$ is exponentially distributed with firing rate 1/HostDemand, it will also release the execution host. The labels $host_acq$ and $host_rel$ are generic for all the translations in this subsection. Algorithm 8.1 will change the word $host$ for the actual name of the host that executes this step. At this moment this information is not known because it is not stored in the Step.

- A Step with Probability is translated as in Figure 8.2(d), which reflects the two flows that the system can follow, the one that really executes the step (right part) and the one that avoids it (left part). There is a special case, the Step with Probability that is preceded by a Branch connector, indeed all the successor Steps of a Branch have a probability of execution. In this case, pattern in Figure 8.2(c) is applied and this translation will be composed with the one of the Branch as given in Figure 8.4(e).

- Pattern in Figure 8.2(e) gives the translation for Steps with RepetitionCount. The RepetitionCount is simply the mean number of times the step repeats when executed. The probabilities for transitions $t5$ and $t6$ are *p* and 1-*p* respectively, where $p = RepCount/(1 + RepCount)$.

- When the Step has both, Probability and RepetitionCount, then it mixes the two previous translations. Figure 8.2(f) depicts the solution to mix (d) and (e), the mix of (c) and (e) is a trivial one.

- Sometimes a Step is refined by a sub-scenario, as expressed by the association between the Step and the Scenario in Figure 8.1. This means that the engineer will refine the specification of the Step. In this case, the Step does not declare a HostDemand since it is implicitly aggregated by the demands of the steps in the sub-scenario. Figure 8.2(g) depicts the translation: place $p1$ will be composed by algorithm 8.1 (later described in Section 8.4) with the starting point of the subnet of the sub-scenario and transition $t1$ with the ending of the scenario, hence the sub-scenario will be integrated into this subnet. Figure 8.2(h) depicts the most complex situation, i.e., the Step that represents a sub-scenario with RepetitionCount and Probability. The cases of a sub-scenario Step with RepetitionCount or with Probability are particular cases of (h).

8.3.2 Resource translation

Resources involve two main classes of the CSM in Figure 8.1: the GeneralResource for their definition and the ResourceManager for their management, i.e., their acquisition and release. As previously explained, a resource can be active or passive, the active resources can be external operations or processing resources. For the GeneralResource class its attributes are Name and Multiplicity (number of units, e.g., number of buffers). The ActiveResource classes adds the OperationTime attribute. When the Multiplicity of a resource is infinite there is no need for it to be translated, since the resource can be shared at any time by any number of execution steps.

Figure 8.3(a) depicts a part of a CSM model with a passive resource –DB component– and the active one where it executes, the CPU for the DB. The component is explicitly acquired and released by the corresponding ResourceManagers ($ResAcq$ and $ResRel$).

Active and passive resources are translated the same way, as given in Figure 8.3(b,c). They are modeled by: a) a place with as many tokens as the Multiplicity indicates, one in this case, b) an input transition modeling the release of the resource and c) an output transition modeling its acquisition. Labels refer the name of the resource as well as its acquisition and release.

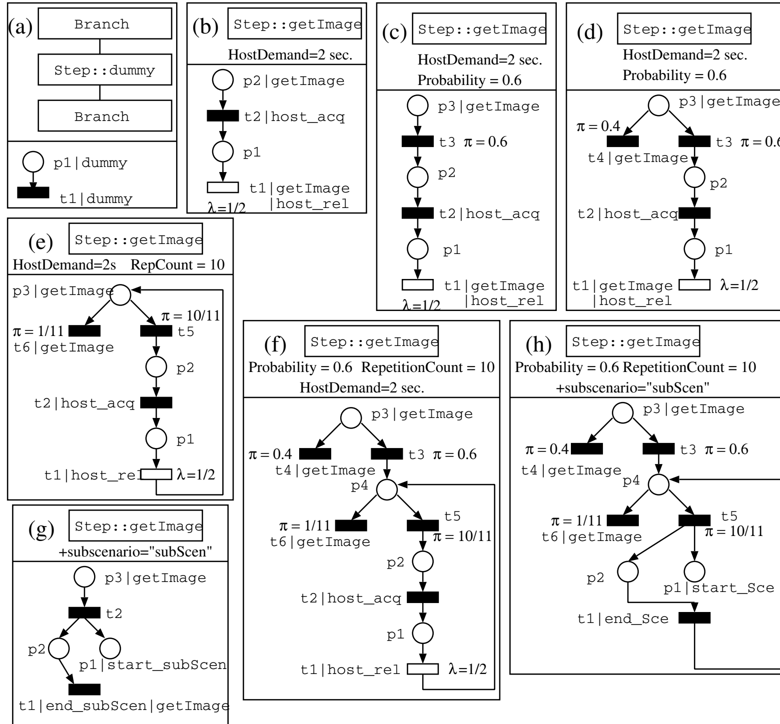


Figure 8.2: Step patterns

On the other hand, the ResourceManager vertices follow the pattern in Figure 8.3(d). Transition *t1* takes its priority from the Priority attribute that represents the execution priority of the component in the host. Labels (DB_acq, DB_rel) are useful for the composition with the passive resource (Figure 8.3(d)).

Figure 8.3(e) composes the subnet in (c) and the subnets in (d) to obtain the final subnet that represents the CSM in part (a).

8.3.3 PathConnections translation

As explained in section 8.2, the path connectors are the means used by the CSM to represent the control flow of the system. Therefore they are meant to explicitly link the steps of the system and their acquisition and release. A path connector can be the source of several VertexOperations and also the target of several of them.

The Sequence pathConnection is graphically represented in the CSM by an arrow connecting two VertexOperations. Figure 8.4(a) depicts its representation in the CSM and also its translation into a GSPN subnet. The subnet is made of a place and a transition labeled

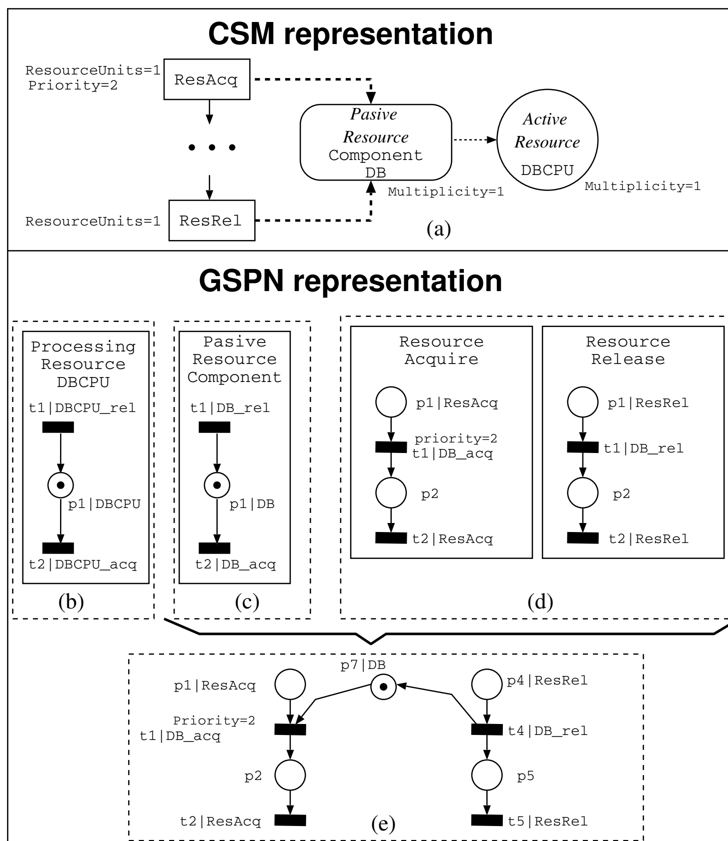


Figure 8.3: GeneralResource and ResourceManager patterns

with the name of the source and target steps. These labels will be useful to merge this subnet with those coming from the steps as proposed in section 8.3.1.

A Fork connector represents the beginning of the parallel execution of different branches in the system that can be eventually connected by a Join. The CSM representation and the translation of these two connectors are given in Figure 8.4(b,c). As in the case of the Sequence, the labels represent the name of the Steps to which eventually these subnets will be merged.

The Branch and the Merge are connectors to represent the start and the end, respectively, of the probabilistic choice of execution in the system. Figure 8.4(e) depicts the translation of the Branch, that is made of a transition and a place labeled with the names of the predecessor and successor Steps. To ease the understanding, we have depicted a translation and composition of the Branch and its precedent and successor Steps (here, $Step_2$ and $Step_3$

are translated as proposed in Figure 8.2(c)). We have also depicted the translation of these Steps following the patterns described in Section 8.3.1.

Finally, in Figure 8.4(d) we can see that the Merge proposes a subnet to connect the alternative execution flows.

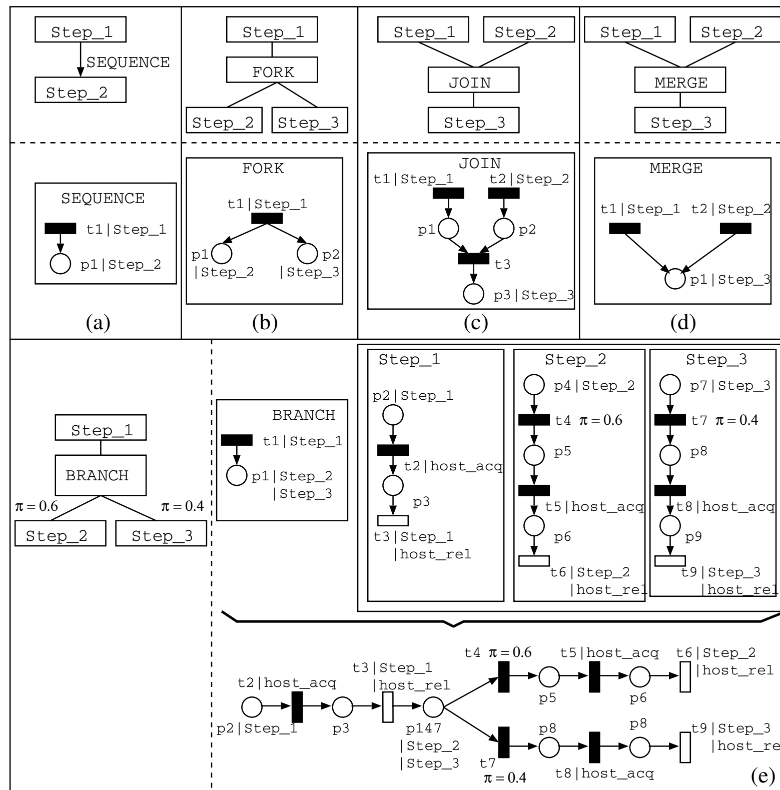


Figure 8.4: PathConnections patterns

Start and End connectors translation

The Start and End connectors are used in the CSM to mark where the model begins and the different ways to finish it.

A Start connector is translated into two places and a transition as given in Figure 8.5(a). The label of place $p2$ will be useful to merge the subnet of the Start with the subnet of its successor step, in this case $Step_1$.

A scenario may own more than one End. When the End is associated with an asynchronous Message, we call it "asynchronous End", otherwise we call it "synchronous End".

Concerning “synchronous Ends”, they can play two roles: either they belong to a subscenario or to a scenario with its own ClosedWorkload. The scenario in Figure 8.13(b) owns both: a “synchronous” End, see left hand side, and an “asynchronous” one. Currently, the CSM allows at most one “synchronous End” per scenario, the rest have to be “asynchronous” [PW07].

Both kinds of “synchronous End” are translated the same way, as in Figure 8.5(b), and the resulting subnet is composed with the subnet of the predecessor step. Moreover, this subnet is also composed in the first case with the subnet of the Step it refines, and in the latter case, the subnet will be composed with the subnet of the ClosedWorkload. The translation of the “asynchronous End” is proposed in Figure 8.5(c), this subnet will be composed with the subnet of the predecessor step.

8.3.4 Workload translation

A Start connector may associate a workload, which can be closed or open. A ClosedWorkload element of the CSM is translated into a place and a transition, as given in Figure 8.6(a). The attribute Population indicates the maximum number of concurrent executions in the scenario. The Population is represented, in the subnet, by the number of tokens in *p1*. The subnet of the ClosedWorkload has to be merged with the subnet of the Start and with the subnet of the “synchronous End”. The resulting PN, see Figure 8.6(a), is a cyclic net. This kind of nets is useful to perform system analysis in steady state.

On the other hand, when a Start connector is attached to an OpenWorkload, it models the distribution function for the scenario arrival events. Among potential arrival patterns, we have implemented in the tool some commonly used: Poisson, periodic and a class of phase type.

Poisson distributions are modeled using an exponential transition with parameter λ, where λ characterizes the arrival rate. Figure 8.6(b) depicts the translation of the OpenWorkload and the resulting Petri net when it is merged with the Start connector.

Workloads whose arrivals are periodic (i.e. the arrival of events follows a constant pattern) are translated as depicted in Figure8.6(c). The difference between this periodic patterns and previous Poisson distribution is that it is used a deterministic transition instead of an exponential one. In Figure8.6(c), trasition *t2* is deterministic and has constant firing time. Therefore, this net is not a GSPN but a Deterministic Stochastic Petri Net (DSPN) [AMC87]. However, we can analyze DSPNs using analysis techniques based on those ones for GSPN as long as the DSPN has at most one deterministic transition enabled [Bal98].

Phase type distributions consist of a set of phases with a given execution time and an absorbing state with a probability for each phase to enter in it. Our tool, instead of implementing the general phase-type distribution, implements a subtype of it, the generalized Coxian. This distribution can be approximated to any probability distribution function changing the number of phases, the mean time in each one and the probability of moving among them. On the other hand, an increment in the number of phases of a Coxian distribution means an exponential increment in the number of possible states; so, this could lead to a large Petri net with a state space explosion problem preventing its performance analysis. Figure 8.6(d) shows the solution we implemented, which is an extension of that in chapter 7 of [AMBC+95]. The

parameter of this function must be made of 4 vectors, each one with as many components as phases: one with the starting probabilities for each phase; one with the moving probabilities to the previous phase; one with the moving probabilities to the next phase and the last with the probabilities of each phase to move to the absorbing state. The sum of ith component of the 2nd, 3rd and 4th vector must be 1.

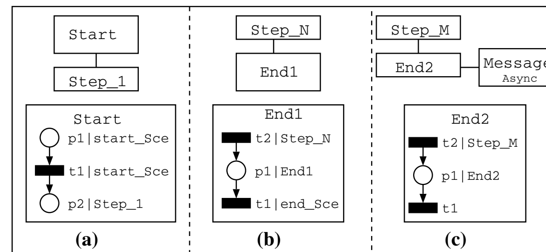


Figure 8.5: Start and End patterns

8.4 Tool development

The CSM to GSPN model transformation patterns proposed in previous section have been implemented in a software tool. This section describes the main points of its development. The tool can be downloaded from [CSM], where a guide about how to use it is also available.

8.4.1 Tool design

Concerning the approach adopted to develop the tool, we studied several choices and their trade-offs. Among them, an interesting one proposed to create an XSLT stylesheet [XSL] to perform an XSL transformation of the CSM into an XML [XML] based PN standard format [BCvH+03]. We ruled out this choice since we were not completely convinced about the current applicability of the PN standard, concretely regarding the stochastic extension and the GSPNs. Finally, we decided to implement, using Java [JAVa], the patterns in section 8.3. This implied to transform the XML representation of the CSM into an application program interface (API) representing GSPNs. The API is the set of java classes in Figure 8.7(a).

This API proposes the abstract class PetriNet as an aggregation of the classes Transition, Arc and Place, abstract as well. The PetriNet class is specialized into concrete classes that will represent the actual PN, e.g., the GSPNPetriNet class or the DSPNPetriNet class.

The GSPNPetriNet class aggregates the places, arcs and transitions of the PNs actually created by the tool. The attributes of these classes represent all the features of our translation (e.g., probability in the transitions or tokens in the places) and graphical information as well. Table 8.2 lists some of these attributes. GraphicElement has the attributes that define the position in a canvas of the Petri net elements. In the Arc class, the attribute toTransition specifies whether the arc targets a transition or a place. In the Place class, initialTokens means

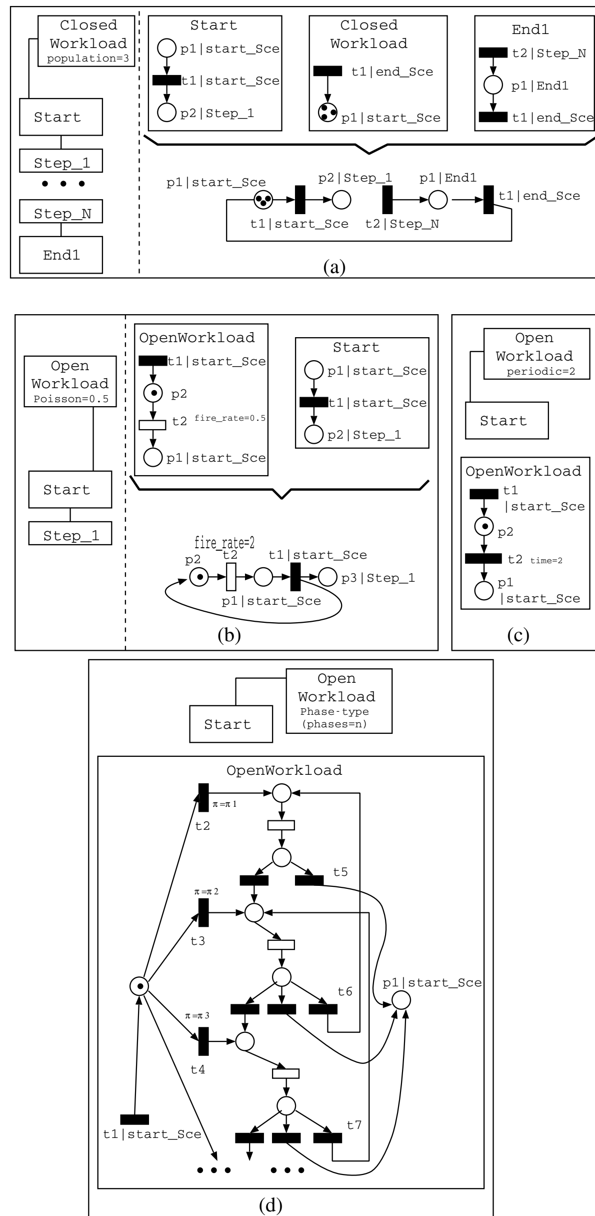
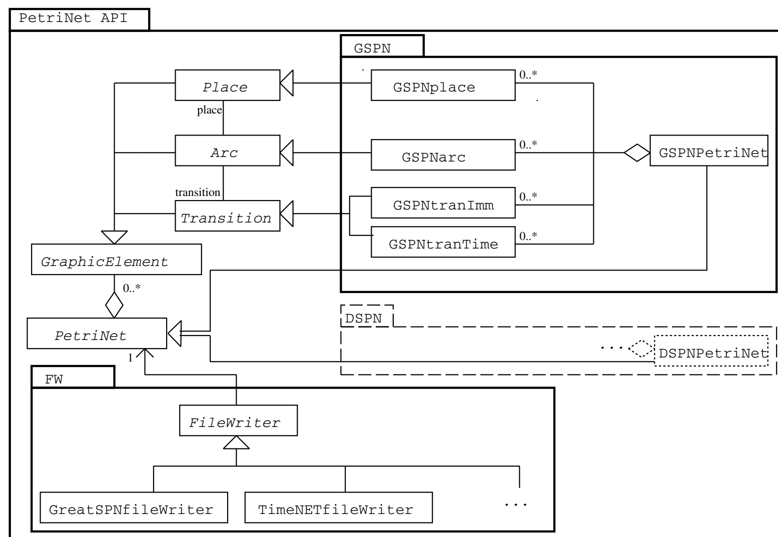
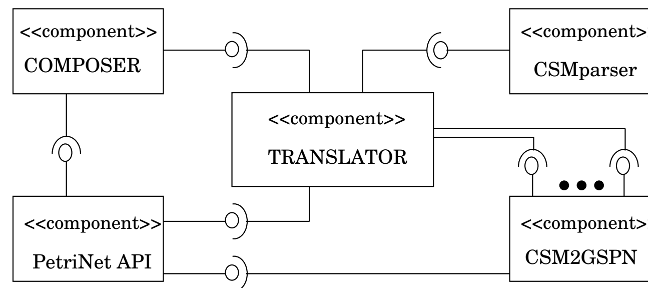


Figure 8.6: Workload patterns



(a)



(b)

Figure 8.7: Tool modules

the number of tokens that our translation sets in the place. The attributes in the GSPNarc class define the common characteristics of an arc in a GSPN. When the attribute inhibitor is `true`, then the attribute toTransition has no meaning. Regarding transitions, there are two classes: immediate and with exponential delay. GSPNtranImm has probability and priority of fire. GSPNtranTime owns the firing rate of the transition. The GSPN classes that represent places or transitions have the attribute labels[] to represent the labels used by the composition operator.

The PN internal representation above described was considered useful for:

- enabling the tool to write the GSPN in different formats, XML-based or not. Currently our tool produces GreatSPN [Gre] and TimeNET [ZFGH00] tool formats, thanks to

the classes that specialize the FileWriter, see Figure 8.7(a). As a consequence, the tool allows to analyze and simulate the GSPN with a variety of techniques, indeed all those implemented by the referred tools.

- future development of particular GSPN analysis or simulation techniques.
- producing other kinds of Petri nets, and accompanied them by some evaluation techniques

Class	**Attributes**
GraphicElement (abstract)	name, info, namePosition, infoPosition, position
Arc (abstract)	*Same as GraphicElement* + toTransition
Place (abstract)	*Same as GraphicElement* + initialTokens
GSPNplace	*Same as Place* + labels[]
GSPNarc	*Same as Arc* + weight, inhibitor
GSPNtranImm	*Same as Transition* + probability, priority, labels[]
GSPNtranTime	*Same as Transition* + rate, labels[]

Table 8.2: Relevant attributes of some classes of the PN API

Figure 8.7(b) proposes the tool high-level design, which is made of five reusable modules. The Translator module is the interface with the user, then receiving a CSM model -in XML format- and providing a GSPN in the desired format. This module implements the Algorithm 8.1, later described.

The CSM2GSPN module implements the functions described in Section 8.3. Then, for each CSM element, the CSM2GSPN offers a method that translates it into the corresponding GSPN (e.g., step2GSPN(), fork2GSPN(), branch2GSPN()). The Translator will call these methods appropriately in Algorithm 8.1.

The design of the CSM2GSPN module features one class, called Csm2Gspn. This class is made just of class methods, so all declared static as java requires. They are those previously mentioned. Moreover, this class cannot be instantiated, so its constructor is declared private. This programming technique is not new, for example, the well-known java.lang.Math class [javb] follows the same pattern. Indeed, having a well-defined set of translation functions, those in Section 8.3, we thought that this module should behave like a mathematical library. Moreover, the Csm2Gspn class ensures that the names of the places and transitions of the GSPNs are not repeated, this entails to keep a slight sense of state. This fact also happens for example in java.lang.Math for the method random(), which stores whether random() has been already called or not. This uniqueness in places and transitions names is also helped by the ID attribute, see Table 8.1, of the CSM element to translate.

The Composer module implements the composition operator described in Appendix B. This module receives as input two GSPNs and two sets of labels (one for places composi-

tion and the other for transitions composition). Then, it produces a GSPN that is the result of the composition operator over the input nets. Different versions of the composition operation have been already implemented by other authors, for example the "algebra" package [BDH01] of the GreatSPN tool. "algebra" performs the composition also for stochastic well-formed nets (SWN). Nevertheless, we have decided to implement our composition module. In fact we need a "composition" not bound to a concrete tool, as it happens to "algebra". Otherwise, we should produce nets in the GreatSPN format only. Then, the analysis programs of the TimeNET tool could not be used, for example. This decision also eases the reuse of the tool and it integration in other architectures (e.g., PUMA [PUMb, PUMa]) since it does not require external functionalities.

The CSM2GSPN and the Composer modules are coordinated by the Translator. The coordination means that they are alternatively called: First the CSM2GSPN to translate the current CSM element and then the Composer to merge the resulting net with the "target net", net that will eventually represent the whole CSM. Algorithm 8.1 concentrates on this coordination.

The CSMparser helps the Translator in managing the XML file with the CSM information, it parses this file using Document Object Model (DOM) trees. The shape of the DOM trees of a CSM is similar to the one depicted in Figure 8.8, this DOM is compliant with the class diagram in Figure 8.1. The root is a CSM element, while the children element are Scenarios and GeneralResource subclasses. Each Scenario keeps a list of children, which store the information about its VertexOperation and PathConnection subclasses. Some of these child elements can in turn have a child list. These are the elements that store the information about a CSM Step which are refined in a sub-scenario. Due to the Composer module, the CSMparser does not need to search the tree for the logical order of VertexOperations and *PathConnections*, it just needs to traverse the DOM tree in a deep-first pre-order form. Then, for the translation of a complete scenario, where $|VertexOperation| + |PathConnection| = N$, the CSMparser does not need to perform N searches for elements in the Scenario subtree, it traverses the subtree just once.

The last module is the API, whose internal structure in Figure 8.7(a) was discussed previously. The API owns the methods for managing places, transitions, arcs and labels of the GSPNs created by the CSM2GSPN and the Composer modules. Thanks to the API, the rest of the modules of the architecture can be reused for the translation of other kinds of Petri nets than GSPN. Actually, the only challenge for reuse is to program a component that substitutes the CSM2GSPN module. This is not really a big effort since the translation in Section 8.3 for GSPN is largely reusable for other kinds of PN (e.g., DSPN or Stochastic Well-formed Nets).

8.4.2 The algorithm for the translation

In Section 8.3 we presented a subnet pattern for each CSM meta-class. Some of these patterns were obtained by composing subnets, it was for example the case for the pattern of the Branch. To this end, we proposed that each place and transition of the subnet were labeled, then the composition would be carried out by merging places and/or transitions with equal labels. By doing so, we actually converted the GSPN into a multi-labeled GSPN (ML-GSPN). The concept of merging subnets is conceptually easy to practice, and it was formal-

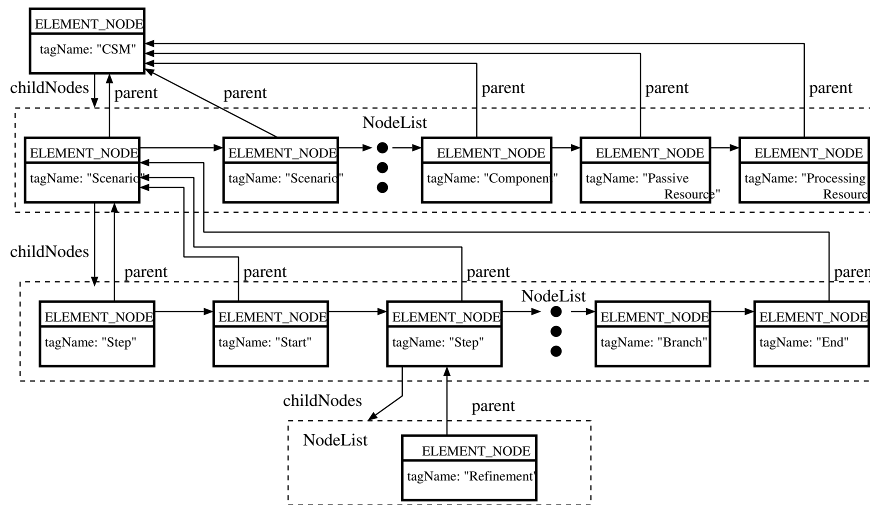


Figure 8.8: Example of DOM tree from a CSM file

ized in [DF96, Ber03] for the case of LGSPN, i.e., GSPN with only one label. Though strictly not necessary for the development of the tool, we have followed this rigorous practice and Appendix B presents the formalization for the case of MLGSPN.

The algorithm starts initializing two variables that hold intermediate GSPNs: partial_mlgspn, that represents at any moment of the execution the elements of the CSM already translated; and list_of_Ends_mlgspns. Next, the algorithm features four parts. The first one (lines 3..12) traverses each scenario in the CSM model to obtain and compose some of the subnets proposed in Section 8.3. The second part (lines 13..15) composes the subnets that represent End connectors (list_of_Ends_mlgspns). The third part (lines 16..19) deals with the passive resources, while the last one (lines 21..24) is dedicated to the active ones. We can devise a simple and structured algorithm because of the effort we previously did:

- **E1**. As expected, the algorithm is absolutely aligned with the design proposed in Section 8.4.1. Thus, the functions `CSM2GSPN` and `Compose` implement what modules CSM2GSPN and Composer conceptualize, respectively.

- **E2**. We conceived the subnets in Section 8.3 to actually ease this algorithm, these nets were designed so that once merged, the desired Petri net is obtained, regardless of the order in which they are composed.

In the first part of the algorithm, the fact of traversing each scenario just once is interesting because it allows to translate all children of a Scenario sequentially, which means not to care about the logical execution order of Steps in the system. Therefore, the traversal was implemented using a simple Iterator pattern[GHJV95] over the DOM tree of the CSM.

This first part deals with the VertexOperations (computational steps and acquisition and release of resources), the Workload, and the PathConnections (the start of the scenario and the

Algorithm 8.1 Translation of a CSM.

Require: An XML CSM file that models a System
Ensure: MLGSPN that represents this System

```
1: partial_mlgspn ← empty_mlgspn()
2: list_of_Ends_mlgspns ← empty_list()
3: for all Scenarios do
4:   for all element ∈ (VertexOperation ∨ Workload ∨
     PathConnection) do
5:     subnet ← CSM2GSPN(element)
6:     if (element.getTagName()==End) then
7:       list_of_Ends_mlgspns ← add(list_of_Ends_mlgspns,subnet)
8:     else
9:       partial_mlgspn ← Compose(subnet,partial_mlgspn)
10:    end if
11:  end for
12: end for
13: for all end_mlgspn ∈ list_of_Ends_mlgspns do
14:   partial_mlgspn ← Compose(end_mlgspn,partial_mlgspn)
15: end for
16: for all element ∈ PassiveResource do
17:   subnet ← CSM2GSPN(element)
18:   partial_mlgspn ← Compose(subnet,partial_mlgspn)
19: end for
20: add-host-characteristics-to-steps(partial_mlgspn)
21: for all element ∈ ActiveResource do
22:   subnet ← CSM2GSPN(element)
23:   partial_mlgspn ← Compose(subnet,partial_mlgspn)
24: end for
25: return partial_mlgspn
```

different control flows). In line 5, it is translated each element into its corresponding GSPN according to the proposal in Section 8.3 by calling the CSM2GSPN module. Actually, all these elements constitute the structure of the scenario and only the resources -which are not part of Scenarios- are not considered yet. After the translation of an element, the algorithm checks whether it is an End (lines 6..10). Then the subnets for the Ends are simply stored in list_of_Ends_mlgspns, while the rest of subnets are composed with partial_mlgspn. The reason of delaying the composition of End elements is explained later.

The composition in line 9 has a particular. When the subnet and the partial_mlgspn have no label in common, then the algorithm produces a new partial_mlgspn that includes the subnet "in parallel", as given in Appendix B. At the end of this part of the algorithm, we have a Petri net structure that links all the computational steps, according to the established control flow, with the subnet of the Workload and the Start as proposed in Figure 8.6. The subnets for the acquisition and release of the resources, Figure 8.3(d), are also linked in this Petri net

to its predecessor and successor steps, however they still need to be linked to the subnets of the resources they manage, as in Figure 8.3(e).

The second part of the algorithm carries out the composition of the End connector subnets with the partial_mlgspn. If we composed the Ends and the partial_mlgspn in the first part of the algorithm, then partial_mlgspn would be wrong, concretely in the case of sub-scenarios which refine more than one Step. Figure 8.9(a) depicts the translation and "too-early" composition of a sub-scenario that is called by two different Steps. We observe that the result is that the sub-scenario cannot return the execution flow to the requester scenario. However, Figure 8.9(b) depicts the translation obtained by delaying the composition of the Ends, which is the expected one for the system. In fact, this was the only case in which we could not find a solution to meet **E2**, so we had to delay the composition.

After the execution of this second part, partial_mlgspn represents everything in the system except the resources. This is so because we are translating scenario by scenario. Not being part of a concrete scenario, the resources belong to the root CSM -as shown in the metamodel in Figure 8.1-, their position in the DOM tree is illustrated in Figure 8.8. Therefore, the third part of the algorithm translates passive resources and compose them with the partial_mlgspn.

The last part of the algorithm (lines 21..24) is dedicated to the translation and composition of the active resources. But prior to this task, it is called the function add-host-characteristics-to-steps (line 20). Considering that each Step executes on a processing resource, this function is responsible for discovering this target resource following the "scope rules" given in [PW07] and substitutes the generic labels $host_acq$ and $host_rel$ (recall Figure 8.2) by the actual name of the host. Thanks to this, the algorithm correctly merges each active resource with the net in this last part. At the end of the algorithm, partial_mlgspn already represents the behavior of the whole CSM.

Although the tool checks the input CSM for some inconsistencies, it is worth noting that we ideally assume a "well-formed" CSM, otherwise the tool could produce an incorrect GSPN. For example, when entering a Fork, the n units of the previously acquired passive resource may be used by at most n threads of the Fork and they will be eventually released. Figure 8.10(a) depicts an erroneous CSM since 1 unit is acquired but 3 released. Figure 8.10(b) depicts the erroneous Petri net that the tool will produce.

8.4.3 Remarks on the analysis of the resulting GSPN

The subnets obtained in Section 8.3 are GSPNs free of choices (formally speaking, they are state-machine or marked-graph PNs). However, when they are composed by Algorithm 8.1, the free-choice property is no longer kept. Therefore, the modeling power of the final GSPN is not restricted to free-choice systems, it can also model resource sharing and competition (formally speaking, the type of the GSPN produced by Algorithm 8.1 is a simple-net). This can be observed in Figure 8.15 where transitions `t92|BufferManager_acq` and `t59|BufferManager_acq` work at the same time as synchronization and choice regarding place `P78|BufferManager`.

Regarding *boundedness*, the final GSPN will be unbounded when there exist at least one open workload. For bounded GSPNs it can be applied classical exact solution techniques based on analyzing the underlying continuous time Markov chain (CTMC). For unbounded

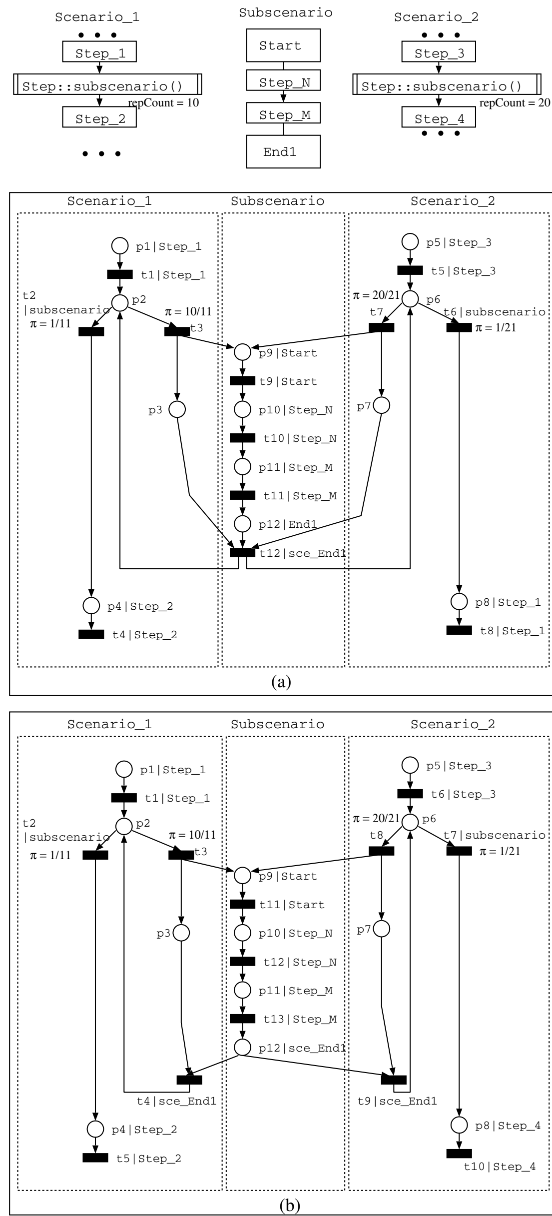


Figure 8.9: Different End compositions

GSPNs, the state space is not finite and consequently its underlying CTMC neither. In this case, depending on the structure of the GSPN, matrix geometric techniques[Hav95] or performance bounds[CS93] can be successfully applied for performance analysis. In cases where the GSPN cannot be analyzed using the previous techniques, simulation may be a good choice. For a comprehensive boundedness characterization of the GSPN produced by the tool, we can say that the net will be bounded if and only if one of the following situations arise:

1. all the workloads are closed and there is no Fork with asynchronous paths.
2. all the workloads are closed and all the elements in the asynchronous paths of the Forks (i.e., the paths that finish in asynchronous Ends) are in the scope of some PassiveResource with multiplicity less than infinite.

Formally speaking, if any of the previous situations happens, all the places will be covered by a P-invariant, and then the Petri net will be bounded.

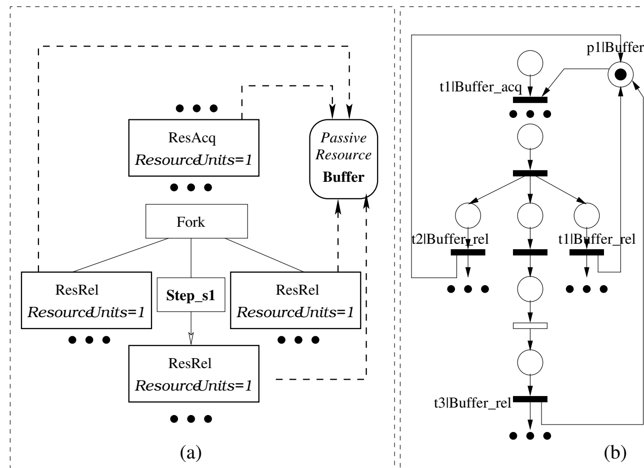


Figure 8.10: Erroneous modeling and translation of a Fork

8.5 Example of system analysis

This section applies the proposed translation to the models of a system with two scenarios. The system is a secure building and has been taken from [XWP03]. The first scenario represents the acquisition and storing of the building video images (A/S-V), Figures 8.11 and 8.13 depict its models: sequence diagram (SD) and the CSM, respectively. The second scenario represents the access control (AC) of the secure building doors, Figure 8.12 depicts its SD, while the CSM can be found in [PW07]. [XWP03] provides an LQN model of this system,

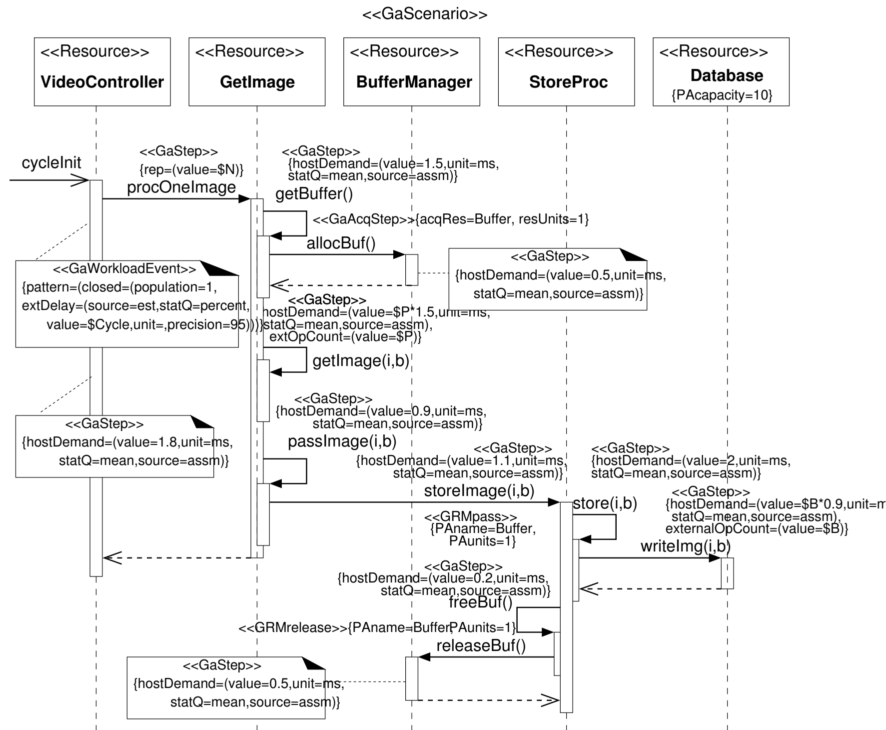


Figure 8.11: Sequence diagram for A-S/V scenario, taken from [XWP03]

moreover the LQN is profiled through several experiments which offer interesting performance results. We will compare these results with ours.

8.5.1 Qualitative properties analysis

From the CSM of the A/S-V scenario we obtained the corresponding GSPN automatically, using our tool. We firstly perform a qualitative analysis of the GSPN, which revealed a deadlock in the procOneImage sub-scenario, Figure 8.13(b). The deadlock was then studied in the UML SD and projected into the CSM as follows. The *Fork* splits the sub-scenario into two execution threads: the long part (sub-scenario asynchronous part) and the short one (synchronous part). The synchronous part cycles to the start of the sub-scenario and acquires the *BufferManager* again but has to wait for the *Buffer*, which could be still being used by the asynchronous one. In turn, the asynchronous part tries to acquire the *BufferManager*, then leading to a deadlock caused by a circular wait.

To evade the problem, the acquisition of resources was swapped in the CSM, see Fig-

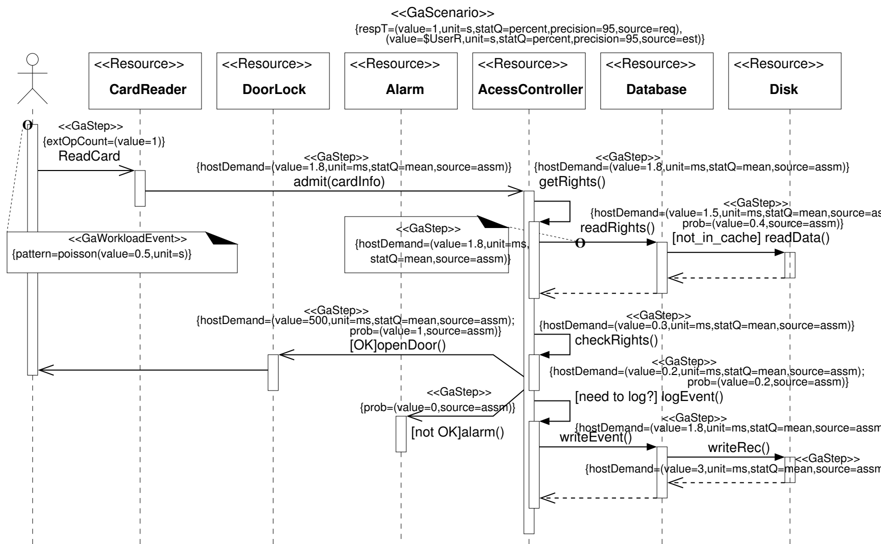


Figure 8.12: Sequence diagram for AC scenario, taken from [XWP03]

ure 8.14. The GSPN was again generated automatically, from the new CSM, see Figure8.15[1]. The new analysis showed that the GSPN was deadlock free.

In the LQN paradigm, the one applied in [XWP03], when a resource is acquired, its layer cannot be traversed again until the execution flow returns back to a previous layer. Then, the layer of the *BufferManager* cannot be traversed back (and the resource released) before its next layer is completed. To avoid this situation, in [XWP03], the *BufferManager* is duplicated, which solves the problem of the not nested operations with resources. But in our opinion, it creates a new one, the problem is that if the scenario could reach a deadlock then it would not be realized with the LQN. As a conclusion, the use of the LQN paradigm for performance analysis is motivated when nested services prevail in the system.

8.5.2 Quantitative properties analysis

Once the GSPN has been shown to own good qualitative properties (deadlock free and liveness), we carry out similar experiments to the ones presented in [XWP03]. Tables 8.3 to 8.6 offer results about: cycle time in seconds for polling all the cameras in the A/S-V scenario; response time (RT) in seconds for a human user accessing the door in the AC scenario; and the normalized utilization of the resources shared by both scenarios. The normalized utilization means the ratio of the mean number of busy resources out of the total number of the corresponding resources. The results in the Tables have been obtained computing, in the GSPN in Figure 8.15, the throughputs of some transitions and the mean number of tokens in

[1]To avoid cluttering, the GSPN in Fig 8.15 appears without the arcs between ProcessingResources and Steps.

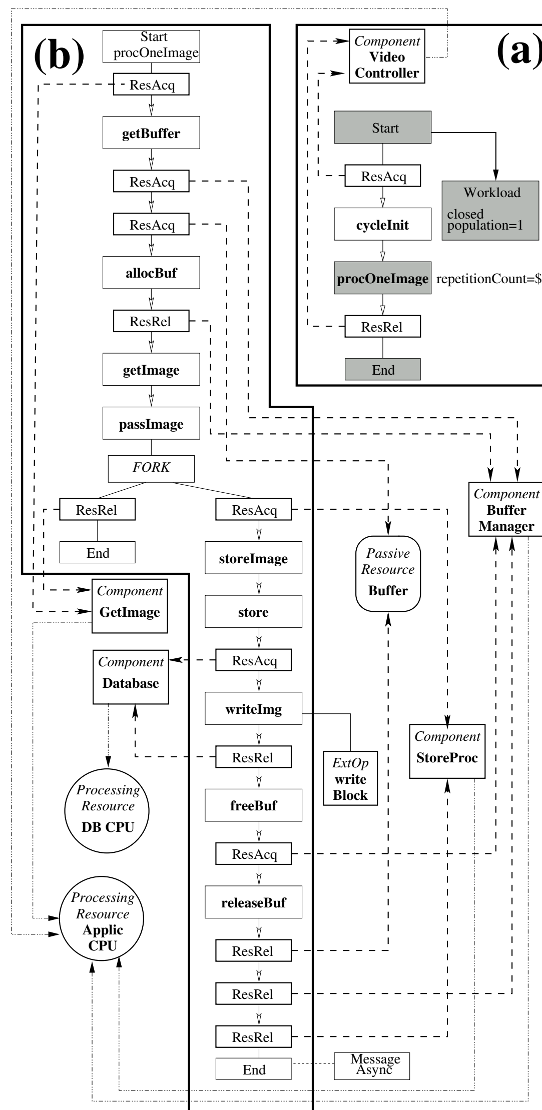


Figure 8.13: CSMs: (a) A-S/V scenario and (b) procOneImage sub-scenario. Taken from [PW07]

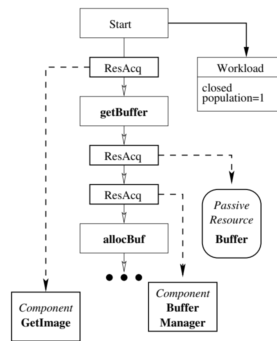


Figure 8.14: The A/S-V scenario without deadlock

some places:

- The Cycle time is computed as the inverse of the throughput of transition `t_END`.
- Using Little's law, the response time (RT) of AC scenario is equal to $\frac{Mean\ number\ of\ tokens\ in\ P_InExe}{Throughput(T_Arrival)}$
- The Normalized resource utilization of X,
  where $X \in \{$GetImage,Buffer,StProc,AppCPU$\}$,
  is computed as $\frac{Number\ of\ X\ resources - Mean\ number\ of\ tokens\ in\ X}{Number\ of\ X\ resources}$

Table 8.3 presents the results when there exists a unique instance of each resource. Experiments considered 10, 20, 30 and 40 cameras in the system. *GetImage* and *Buffer* components show to be the critical resources, they are busy almost all the time. The other resources are not affected by the number of cameras. The A/S-V cycle time shows an increment proportional to the number of cameras, while the RT is not affected. The purpose of the next studies is to decrease the A/S-V cycle time for 40 cameras without increasing the AC response time.

Table 8.4 gives the results when the number of *Buffers* is increased. The A/S-V cycle time decreases only when the number of *Buffers* changes from 1 to 2, this is because the "synchronous and asynchronous parts" in the A/S-V can now work concurrently. However, new increments in the number of *Buffers* do not increment the system concurrency, then the A/S-V cycle time is not decreased. The utilization of the *Buffer*, which is always saturated, confirmed this. However this resource is not the bottleneck since its increments do not decrease the A/S-V cycle time. So, the bottleneck has to be either the *StoreProc* or *GetImage* components, which are fully saturated too. Actually, the bottleneck is the *StoreProc* because the *GetImage* only works in the synchronous part and can never exist concurrent executions of the synchronous part. This conclusion can be reached when simulating the GSPN, then counting the mean number of tokens in place P69 in Figure 8.15, that is 0. This bottleneck in *StoreProc* forces the execution of the asynchronous part to be sequential.

Our next study (Table 8.5) gives the results when there are 40 cameras, 4 *Buffers* and the number of *StoreProc* is increased. When a new *StoreProc* is added, then the asynchronous

Figure 8.15: GSPN for *AC* and *A/S-V* scenarios

(1)	Average time		Normalized resource utilization			
	Cycle	RT	GetImage	Buffer	StProc.	AppCPU
10	0.328	0.187	0.992	0.9992	0.590	0.569
20	0.656	0.186	0.998	0.9996	0.590	0.569
30	0.985	0.182	0.997	0.9997	0.590	0.569
40	1.316	0.183	0.998	0.9998	0.592	0.570

(1) Number of cameras.

Table 8.3: GSPN results

part will allow concurrent executions, then decreasing the A/S-V cycle time. Before the replication of the *StoreProc*, the *Buffer* was almost fully saturated, because the synchronous part had to wait for an empty *Buffer*. After the replication, the *Buffers* are released faster and the synchronous part does not have to wait so long for a free *Buffer*. It is worth noting that now the utilization of the *AppCPU* has increased, what explains the slight increment in the AC user RT. When the *StoreProc* changes to 3 units, the A/S-V cycle time does not decrease anymore, so, the *StoreProc* is no longer the bottleneck. Therefore, to improve the system performance with the current design, the unique option already available is to replicate the *AppCPU*, which actually is saturated. Table 8.6 gives the results of these experiments. When an *AppCPU* is added, both RT and A/S-V cycle time decrease their estimated values.

(2)	Average time		Normalized resource utilization			
	Cycle	RT	GetImage	Buffer	StProc.	AppCPU
1	1.316	0.183	0.998	0.9998	0.592	0.570
2	1.130	0.198	0.998	0.9940	0.998	0.675
3	1.117	0.188	0.998	0.996	0.999	0.672
4	1.100	0.186	0.998	0.997	1.0	0.673
7	1.110	0.190	0.998	0.998	1.0	0.673
10	1.111	0.178	0.998	0.998	1.0	0.673

(2) Number of Buffers.

Table 8.4: GSPN results: number of Buffers

(3)	**Average time**		**Normalized resource utilization**			
	Cycle	RT	GetImage	Buffer	StProc.	AppCPU
1	1.100	0.186	0.998	0.997	1.0	0.673
2	0.798	0.198	0.997	0.830	0.940	0.947
3	0.756	0.189	0.997	0.690	0.651	0.986

(3) Number of StoreProc.

Table 8.5: GSPN results: number of StoreProc threads

(4)	**Average time**		**Normalized resource utilization**			
	Cycle	RT	GetImage	Buffer	StProc.	AppCPU
1	0.798	0.198	0.997	0.830	0.940	0.947
2	0.640	0.171	0.997	0.620	0.665	0.589
3	0.645	0.173	0.997	0.615	0.652	0.393

(4) Number of AppCPU.

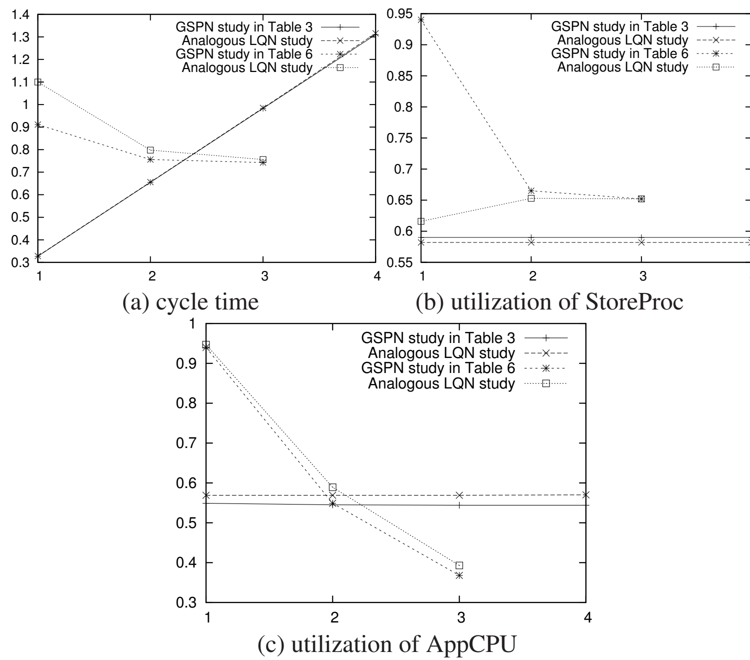
Table 8.6: GSPN results: number of AppCPU

8.5.3 LQN and GSPN results comparison

When comparing the results obtained here with those in [XWP03], we can see that they are very close, for example, if a resource is saturated in the LQN, then it is in the GSPN analysis too. Therefore, for this case study there is no loss of information in the CSM representation, nor in the CSM-GSPN translation. Then, it seems that the proposed translation and its automatic implementation may work as good as a performance analyzable model that has been directly conceived by a domain expert.

Figure 8.16 shows graphically a comparison between our results and the ones in [XWP03]. Among the results discussed in the previous section, we have chosen a representative subset for comparison: the cycle time of A/S-V scenario, the utilization of the StoreProc and of the AppCPU. In each graph, we represent information in Table 8.3 and in Table 8.6. Table 8.3 is the base case of the study, while Table 8.6 is the most complex case –the one where more resources are replicated–. Of course we compare these two cases with their analogous in [XWP03]. For the base case, the x-axis represents tens of cameras. For the case in Table 8.6, the x-axis represents the number of AppCPUs used.

In graph 8.16(a), for the base case our results are identical to those obtained in [XWP03] using LQNs, indeed you cannot distinguish these two lines in the graph. Also for the base case but in graphs (b) and (c) results are extremely close.



(a) cycle time (b) utilization of StoreProc

(c) utilization of AppCPU

Figure 8.16: Comparison with the results in [XWP03]

Regarding the most complex case, results in (a) and (b) converge for 2 and 3 AppCPUs, however they are different when just one AppCPU is considered. Results in part (c) are very close for GSPNs and LQNs.

Finally and not represented in these graphs, it is worth noting that we have also compared the response time of the Access Control scenario. It has been almost constant for all studies. However, in our study, these response times are around 180 milliseconds, while in [XWP03] are around 130 milliseconds. Even if the requirement was satisfied in both studies, there is a substantial gap between results; maybe not in terms of absolute value (they are 50 milliseconds of difference) but in terms of relative values (our obtained RT is around 38% higher than their one).

8.6 Conclusion

This chapter has shown part of our research work in model-to-model transformations; the part that uses the intermediate model CSM. The contributions of this chapter are: first, we have presented a theory underlaying the tool (CSM-GSPN transformation patterns), preliminary ideas were given in [WPP+05]. Second, we have developed a tool that implements this theory and we have shown the main points of its development. Lastly, we put the tool to work and we applied the model transformation to a real case study from literature. We evaluated

the generated GSPN and we compared the performance estimation obtained with the LQN results in [XWP03].

Chapter 9

Model To-Model Transformations: D-KLAPER and GSPNs

This chapter describes the second part of our research in model-to-model transformations, which is centered in the intermediate language D-KLAPER. Since this language was conceived as resource-oriented, it is specially well-suited for representing software models that are engineered following the component-based paradigm. We first present a slight extension of D-KLAPER metamodel that enables it to create models that represent the reactive behaviors of self-adaptive systems. Later in this chapter, there are presented theories for the model-to-model transformation chain from UML state machine diagrams to the extension of D-KLAPER and from the extended D-KLAPER to GSPNs.

9.1 Motivation

Self-adaptive systems, to the extent that they maintain ongoing interactions with their environment and are able to react to external stimuli, are reactive systems. Using this membership, we describe model transformations for the particular concepts of reactive systems that can be directly used by reactive self-adaptive systems.

Reactive systems cannot be described in terms of a single function that maps inputs to outputs. The response that a reactive system provides to an input event, for example, depends on the current state of the system, which, in turn, is a function of the already received inputs [HP85]. Additionally, the system may offer services or behaviors that can be invoked, also by third parties, in the form of events to which the system reacts. Some examples illustrate them. Real-time systems, after time-outs, awake tasks by sending events, so they react to accomplish their works. An agent may be requested to perform some behavior by means of a call to an event it offers. Service software reacts to incoming calls and manages them to offer adequate responses.

At this regard, we identified three necessary and intrinsic abilities of systems that show reactive behavior:

(a) Their ability to suspend their execution until the eventual reception of a signal or event.

(b) Their ability to accept and manage a signal/event while they are doing some computing work.

(c) Their ability to send signals/events to their components or to another systems for them to react.

Designing and maintaining the functionality of such systems is a challenging task and so are their extra functional properties. In this respect, this chapter describes our contribution to support the design and management of reactive software systems by means of the model-based analysis of their effectiveness, focusing in particular on their ability to meet extra functional and QoS requirements related to performance and dependability attributes.

The idea of exploiting MDD methodologies for QoS assessment has emerged in recent years (see, for example, [AGM08, BKR09, MM06] and papers in [wos10]), but existing MDD-based methodologies for the generation of QoS analysis models did not consider the modeling of adaptable reactive systems yet. Following MDD methodologies and the approach of two-step transformation process from design oriented to analysis oriented models, we propose model-to-model transformations for QoS assesment of reactive systems. As pivot language between transformations we propose the utilization of an extension of D-KLAPER [GMS07a, GMS07b], which is a resource oriented intermediate language.

This chapter builds on and extends results presented in [GMR09, GMS07a, GMS07b]. Specifically, we extend the intermediate modeling language presented there with a new simple feature aimed at modeling the specific aspects of reactive systems. The proposed extension (illustrated in Section 9.2) follows the underlying philosophy of this intermediate language definition maintaining it as minimal as possible.

The chapter is organized as follows. Section 9.2 presents the core concepts of D-KLAPER and the proposed extension. In Section 9.3, we describe the proposed transformation path and we show how this notation can be used to model reactive systems. In Section 9.4, it is shown the practical application of the presented ideas trough a simple example of adaptable reactive system. Finally, Section 9.5 concludes the chapter.

9.2 The D-KLAPER intermediate model

D-KLAPER [GMS07b] is defined as a MOF (Meta-Object Facility) compliant metamodel, where MOF is the metamodeling framework proposed by the Object Management Group (OMG) for the management of models and their transformations within the MDD approach to software development [KWB03]. We point out that D-KLAPER is not intended to be directly used by system designers. Indeed, in the modeling framework its authors envisaged, D-KLAPER plays a role analogous to that played by the bytecode language in a Java environment. Hence, D-KLAPER provides a purposely minimal set of elementary and abstract concepts and notations. More expressive concepts and notations used by system designers to build their models should then be mapped to D-KLAPER, with the support of automatic

model transformation tools. In particular, D-KLAPER is built around these two elementary abstract concepts: (i) a software system (and its underlying platform) can be modeled as a set of resources which offer and require services; (ii) a system change can be modeled by a change in the binding between offered and required services.

Hereafter, we illustrate the main classes of the D-KLAPER metamodel illustrated in Figure 9.1; for a complete description of D-KLAPER the interested reader can refer to [GMR09]. The red line in Figure 9.1 highlights the proposed extension.

Resource and Service metaclasses provide an abstract representation of the software system and the part of its environment consisting of the platform where it is deployed. This representation is based on the consideration that systems are often structured according to a layered architecture, where components at a given layer actually play the role of resources exploited by upper layers; hence, this overall architecture is modeled as a set of Resources which offer Services (services, in turn, may require the services of other resources to carry out their own task). A D-KLAPER Resource is thus an abstract modeling concept used to represent both software components and physical resources like processors and communication links.

Workload metaclass models the part of the system environment consisting of the demand arriving to the system from external users (which may be human beings, or other systems), represented by a set of Workloads.

The metaclasses described above share the *Behavior* metaclass, which provides a common representation for the dynamics of the activities occurring within each submodel. As shown in Figure 9.1, a Behavior is modeled as a directed graph of Steps. Each Step may be a:

- *Activity* step: it models an activity that may take time to be completed, and/or which may fail before completion, thus providing the basic information for performance or dependability analysis. A special kind of Activity is a ServiceCall, which models the request for the service provided by some Resource. A ServiceCall may have Parameters. D-KLAPER Parameters are intended to be an abstraction of the parameters actually used in the service requests addressed to hardware or software resources. For example, a "List" parameter sent to some list processing resource could be abstracted by an integer parameter representing its size, under the assumption that this is the only relevant information for performance analysis purposes. The relationship between a ServiceCall and the actual recipient of the call is represented by means of instances of the Binding metaclass.

- *Control* step: it models transition rules from step to step, like a branch or a fork/join.

- *Reconfiguration* step: it models a basic change operation, corresponding in D-KLAPER to the addition or removal of a Binding between a ServiceCall step and the corresponding Service. Only the behavior associated with a TriggerProcess or an AdaptationService is allowed to contain Reconfiguration steps.

The semantics of a Behavior are similar to that of other behavioral models like Execution Graphs [SW02a] or UML Activity Diagrams [Obj02]. As D-KLAPER is intended to support the stochastic analysis of performance or dependability attributes, timing, failure and

control information associated with steps of a behavior is specified according to a stochastic setting: thus, information like the time to failure or the time to completion of an Activity are defined by suitable random variables; analogously, control information like the selection among alternative transitions, or the number of repetitions of a loop is expressed by suitable probabilities and random variables. D-KLAPER supports the specification of random variables in different ways, ranging from their mean value, to higher order moments, up to the complete distribution. It depends on the target QoS analysis methodology whether this information can be thoroughly exploited (e.g., analytic methodologies for queueing network models usually consider only mean values).

9.2.1 Metamodel extension

The D-KLAPER metamodel has been extended with a new association between *Binding* and *Transition* shown with a red line in Figure 9.1. A D-KLAPER *Transition* establishes the execution order of two or more *Steps*. The new association is introduced to allow a *Step* to wait for the execution of its successor/s, so, now a successor *Step* may not execute immediately after its predecessors complete. Now, a *Step* can wait the execution of a *ServiceCall* associated with the same *Binding* as the *Transition*, then accounting for the first one of the three abilities studied for reactive systems, (showed as (a) in section 9.1 list), i.e., to suspend system execution until the eventual reception of an event represented by the association between *Binding* and *ServiceCall*. Summarizing, the new association allows gaining the ability (c) since the system now can send events, precisely through *ServiceCalls* which, by means of *Bindings*, are bound to *Transitions*. On the other hand, ability (b) is also gained since a *Step* can be interrupted when any of its `out` *Transitions* receives an event through their associated *Binding* due to the occurrence of the bound *ServiceCall*.

Some issues deserve to be clarified:

- A *Transition* without associated *Binding* is "taken" when its precedent *Step* (role `from`) finishes its execution.

- A *Transition* with associated *Binding* is "active" when its `from` *Step* is executing. A *Transition* with associated *Binding* is "taken" when it is active and the *ServiceCall* associated to its same *Binding* is performed. When a *Transition* with associated *Binding* is taken, the system: 1) *interrupts* its `from` *Step*; 2) deactivates itself and all the *Transitions* with the same `from` as it; and 3) activates the `to` *Step*. So, when the *ServiceCall* associated to the *Binding* of an "active" *Transition* takes place, the execution is shifted immediately from its `from` to its `to` *Step*.

- It is worth noting that since each *Binding* can be associated with several *Transitions*, it can happen that these transitions could be concurrently active and waiting for the *ServiceCall* execution to be taken. We have decided that only one of these *Transitions* may be taken (the choice will be non-deterministic). Consider that another alternatives could be taken into account, for example: 1) to allow to fire all *Transitions* associated with the same *Binding*; 2) to perform a probabilistic choice by assigning an attribute "probability" to each *Transition*.

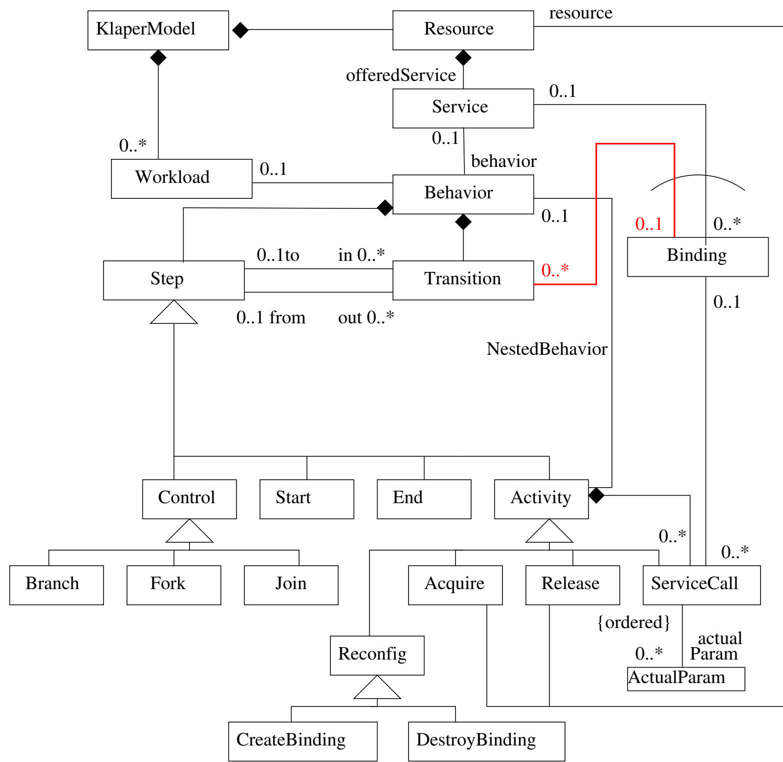


Figure 9.1: D-KLAPER metamodel

We show through four examples, depicted in Figure 9.2, that the proposed extension is able to model the characterizing points of reactive systems introduced in Section 9.1. They show two activities, `activity1` and `activity2` in sequence. In this Figure, the new associations between *Bindings* and *Transition* are depicted as dotted lines.

Part (a) in Figure 9.2 depicts these two activities as a fragment of a workflow with neither interruption nor suspension abilities, i.e., the typical order between activities that was already possible to model with D-KLAPER. Part (b) extends the previous one considering *interruption* of `activity1` due to the reception of a signal (`signal1`). The activation of the outgoing *Transition* from `activity1` which is associated to the *Binding* is made at the same time as the activation of the activity itself. Therefore, if `signal1` is received (i.e. it is executed the *ServiceCall* associated with the same *Binding*) while `activity1` is executing, the *Transition* associated with the *Binding* is taken, then deactivating immediately the execution of `activity1` and starting also instantaneously the execution of `activity2`. In general, the activation of an outgoing *Transition* with associated *Binding* from a *Step* is made at the same time as the activation of the Step itself. Note that the interruption due to

a signal could lead the execution to another *Step* different to `activity2`, but it has been used the same to keep the example simple. If `signal1` is not received, then `activity1` finishes its execution, the *Transition* without associated *Binding* is taken then deactivating both `activity1` and the *Transition* with associated *Binding*, and just after `activity2` is activated and starts its execution.

Part (c) in Figure 9.2 extends part (a) considering *suspension* of `activity2` until an event/signal reception. Suspension is modeled with a *Step* whose outgoing *Transitions* are associated with a *Binding* entity. In this case, `activity1` and `activity2` are connected by means of an intermediate activity called `inSuspension`. Execution of `activity1` is always completed (non interrupted) and later the execution can be suspended until the reception of `signal1`, which is necessary to proceed with `activity2`.

Part (d) in Figure 9.2 extends part (a) by considering the mix of both *interruption* and *suspension*. On the one hand, if `activity1` is in execution and `signal1` is received, the execution is interrupted, the *Transition* is taken and `activity2` is activated to execution. On the other hand, if `activity1` completes but `signal1` has not been received yet, then the execution is suspended until reception of such signal. In short, `activity2` immediately starts its execution if and only if `signal1` is received. Note that part (d) is almost the same as part (b) but removing the transition without associated *Binding* (second link between `Activity1` and `Activity2`) in order to gain the suspension property.

9.3 The model-driven framework for reactive systems

In this section we follow the key points of an MDD-based approach to the generation of a performance/reliability model for a reactive system (explained in section 9.3.1). We first give a short overview of the selected design model (section 9.3.2) and then of the two transformations steps (sections 9.3.3 and 9.3.4) built around D-KLAPER.

9.3.1 The basic methodology

As we mentioned previously, the goal of an intermediate language is splitting the complex task of deriving an analysis model (e.g., a Petri net or a queueing network) from a high level design model (expressed using some design oriented notation) into two separate and presumably simpler steps.

The input of our framework is represented by a design-level model of a reactive system. There exist some non-formal languages and notations that allow to model reactive behavior in software systems. Among them Harel statecharts [Har87] or UML state machines [Obj02]. We select UML state machines as design models since they have become a de facto standard for software reactive behavior specification and we show how the proposed D-KLAPER extension is able to tackle the new design model characteristics. In the next subsection, we briefly recall the UML state machines syntax, and identify how they address the three identified abilities for reactive behavior (suspension, interruption and event sending).

Independently from the selected notation, design models may lack performance and/or reliability information which is necessary to derive meaningful analysis models. Therefore, these models must be annotated with missing information about non-functional attributes.

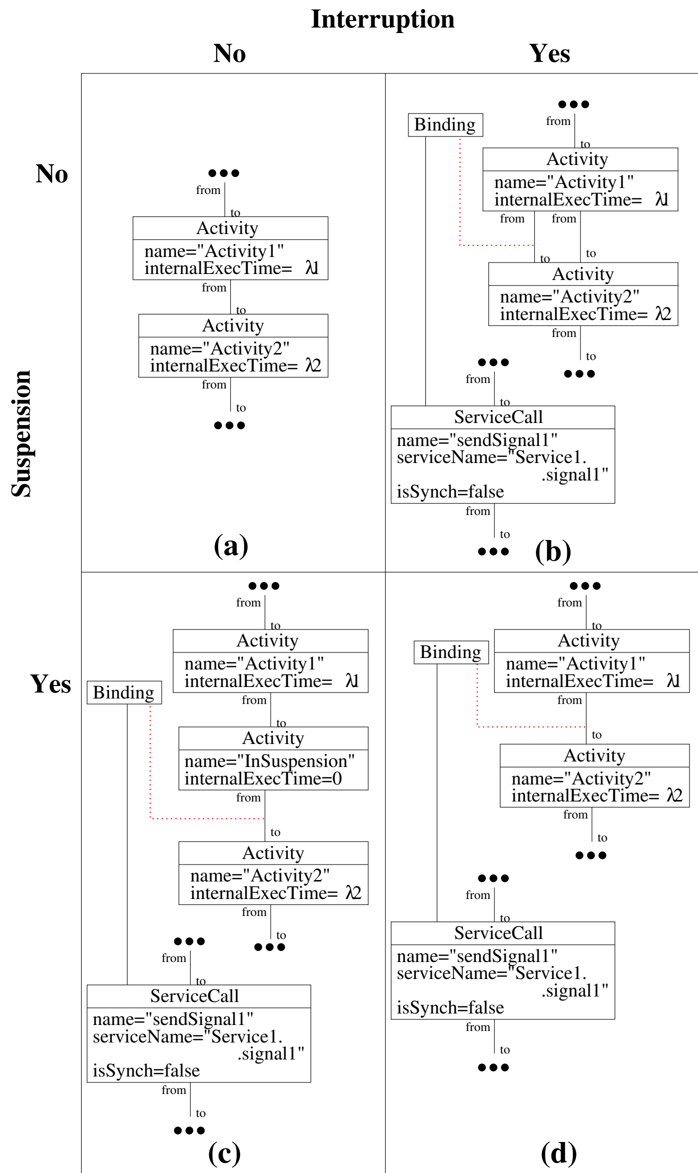


Figure 9.2: D-KLAPER examples for combination of Interruption/Suspension

In the case of UML design models, annotations can be added following the MARTE-DAM profile [BMP11].

At this point, we generate D-KLAPER models starting from the design models with performance/reliability annotations using model-to-model transformations and following the main steps illustrated in [GMS07a].

Finally, we can generate from the D-KLAPER model a performance and/or reliability model expressed in some machine interpretable notation, and then we can solve it using suitable solution methodologies. In our framework, we take advantage from the already defined translations from D-KLAPER into formalisms that can adequately represent reactive behavior, such as Petri nets [PPM10], and we modify and update the translation process to effectively support the reactive property added to D-KLAPER.

The predictions obtained from the analysis of performance and/or reliability models obtained at this step can be exploited to perform *what-if* experiments and to drive design decisions leading to meet the desired quality requirements.

9.3.2 UML state machines as reactive systems models

A UML state machine is made of *states* and *transitions*. There are different kind of states (e.g., pseudostate, simple or composite). States own outgoing transitions that target another states. So *transitions* link states and are made of two parts. The *reactive part* that specifies the event that triggers the transition, and the *proactive part* that specifies the event that will be send. When the *reactive* part is empty, it is called *automatic* and taken as soon as the activity completes its execution. In a state, it can also be specified an *activity*, which is meant to spend some computation time.

A UML state machine can specify the three abilities identified for reactive behavior:

- There can be *sent events* between state machines. That events can be produced among others by the proactive part of a transition.
- *Execution interruption* is modeled by a state that when executing an *activity* accepts an event (obviously an event that can trigger one of its outgoing transitions).
- The execution can be *suspended* in a state, but it is necessary condition that all its outgoing transitions own reactive part, i.e., a trigger event. Assuming that, there are two ways for modeling suspension: (a) If the state has not *activity*, then the execution is suspended upon entrance in the state; (b) If it owns *activity*, then the execution is suspended from the activity termination to the arrival of an event triggering whatever transition.

Note that the purpose of this work is not the comprehensive transformation of every characteristic of UML state machines into D-KLAPER model, but the enhancement of D-KLAPER to deal with reactive systems. Hence, the use of simple UML state machines is enough to show the three studied properties of reactive systems. Herein, it is carried out the transformation of such simple UML state machines; being out of the scope of this work complex state machines, such as those with composite states, concurrent regions, deferred events, or history states.

9.3.3 Transforming UML state machines into D-KLAPER models

In this section we present how a generic simple state of a UML state machine, see Figure 9.3, is translated into a D-KLAPER model, see Figure 9.4. Considering that a UML state machine is an aggregation of states, together with its outgoing transitions, we could easily obtain the D-KLAPER model corresponding to a UML state machine made of simple states.

We describe the translation through Figure 9.4, that obviously follows the execution model of a simple state proposed by UML. First, upon state entrance, the *entry* action has to be executed, so it is converted into a D-KLAPER activity that has an `internalExecTime` equals to zero. Then, the *doActivity* is also converted into a D-KLAPER activity but in this case the `internalExecTime` has to be greater than zero. The translation of an event-driven outgoing transition is more laborious:

- the reactive part is represented in D-KLAPER by a link to a *Binding*.
- the proactive part in UML can be specified either by an *action* or by the *sending* of an event. In D-KLAPER, the former will be obviously translated as the *entry* actions, and the latter with a `ServiceCall`.
- finally, the *exit* action specified in the state is also translated as a part of the transition, see Figure 9.4.

The translations for internal transitions and automatic outgoing transitions are the same as the previous one but considering that:

- the *internal* does not executes the *exit* action.
- the *automatic* has not reactive part.

Finally, it is interesting to remark that:

- wherever an *action* (entry or exit) can be specified, then it can be substituted by the *sending* of an event. In fact, that is what happened in the proactive part of a transition.
- the translation of the *exit* action will appear as many times as outgoing transitions exist in the state (both automatic and event-driven).

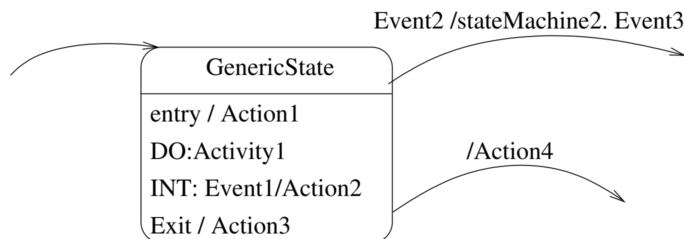


Figure 9.3: UML model of a generic state

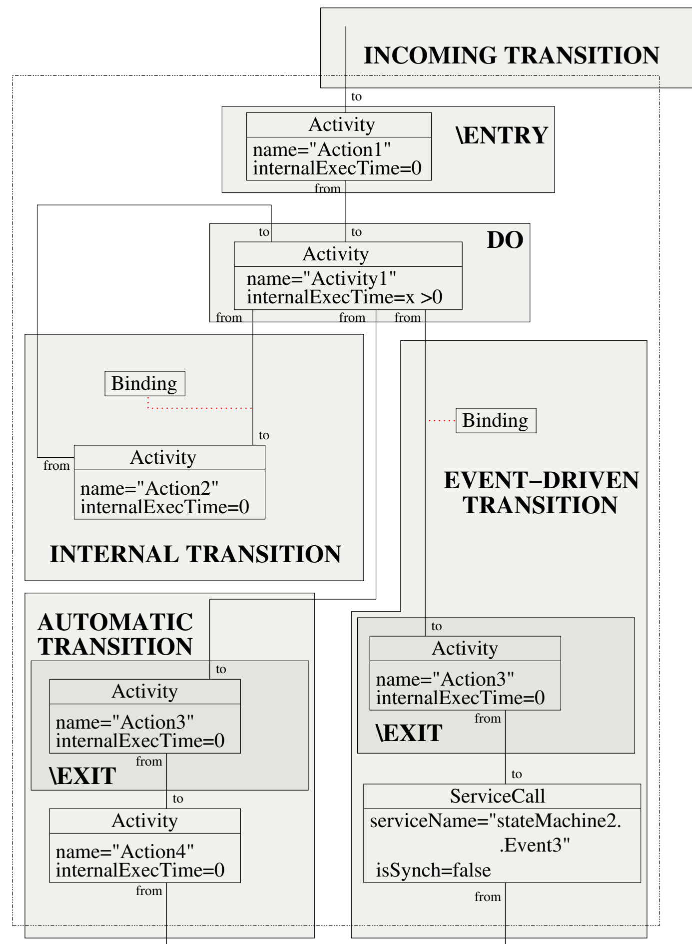


Figure 9.4: D-KLAPER model of a generic state

9.3.4 Transforming D-KLAPER models into Petri nets

D-KLAPER can be transformed to a number of analysis models, for example Deterministic and Stochastic Petri nets (DSPN) [AMC87], that (1) will allow the evaluation of performance and reliability and (2) since they are a formal method, the source model can also gain a representation with formal execution semantics.

A DSPN system is a 8-tuple $\mathcal{S} = (P, T, \Pi, I, O, H, W, M^0)$, where P is the set of places, T is the set of transitions (immediate and timed), $P \cap T = \emptyset$; $\Pi : T \rightarrow \mathbb{N}$ is the priority function that assigns a priority level to each transition. $I, O, H : T \rightarrow 2^P$ are

the input,output, inhibition functions, respectively, that map transitions onto the power set of P; $W : T \rightarrow \mathbb{R}$ is the weight function that assigns rates to exponentially distributed transitions, constant delays to deterministic transitions and weights to immediate transitions. $M^0 : P \rightarrow \mathbb{N}$ is the initial marking of the net.

The Petri nets in Figure 9.5 correspond to the translation of the D-KLAPER in Figure 9.2 which indeed introduced the three abilities we identified for reactive behavior.

The *event sending* ability is represented in parts (b), (c) and (d) in Figure 9.5 by means of `ServiceCall` transitions, which are in charge of create a token in the `Binding` place. The *suspension* ability is represented in parts (c) and (d) by means of $\texttt{Transition}_{Bind}$. In these parts, $\texttt{Transition}_{Bind}$ are the only ones that link `Activity1` with `Activity2` Petri net fragments. Therefore the execution flow cannot reach `Activity2` until the firing of such transitions, which are further waiting for the creation of a token in `Binding` places. *Interruption* ability is represented in parts (b) and (d) through arcs between `Activity1` places and $\texttt{Transition}_{Bind}$. In these parts, if a token in `Binding` place is created while `Activity1` is executing, the token will be removed from the input place of `Activity1`, then interrupting it, and a token wil be created in `Activity2` place. The translation of the rest of D-KLAPER metaclasses, which are not illustrated in this chapter, follows the ideas introduced in [PPM10].

9.4 Example application

In this section, we illustrate reactive behavior with a simple example of a dynamic software system. UML state machines and a sequence diagram describe the behavioral design of the system, they are also extended with a profile, MARTE-DAM [BMP11], that introduces performance and reliability system views. These diagrams are translated into a D-KLAPER model which preserves the desired reactive properties and also accounts for performance and reliability. Finally, the translation of D-KLAPER model into Petri Nets and their analysis help to verify, in early life-cycle stages, whether the system fulfills some performance and cost requirements taking also into account some dependability properties such as availability/reliability.

9.4.1 Structural specification

A UML component diagram extended with MARTE-DAM annotations [BMP11] is depicted in Figure 9.6(a). Component *C1* offers service *S0* and perform calls to *S1*, indeed there are two choices to invoke *S1*:

- a) as an Internet service that comes at a price;
- b) as a COTS component that has already been integrated and executes for free, therefore being the default option.

C1 is made of four classes as detailed in Figure 9.6(b). The *C2* COTS component reliability specification warns about a Mean Time To Failure (MTTF) equal to 10^5 time units (tu), and a Mean Time to Repair (MTTR) of $5 \cdot 10^3$ tu.

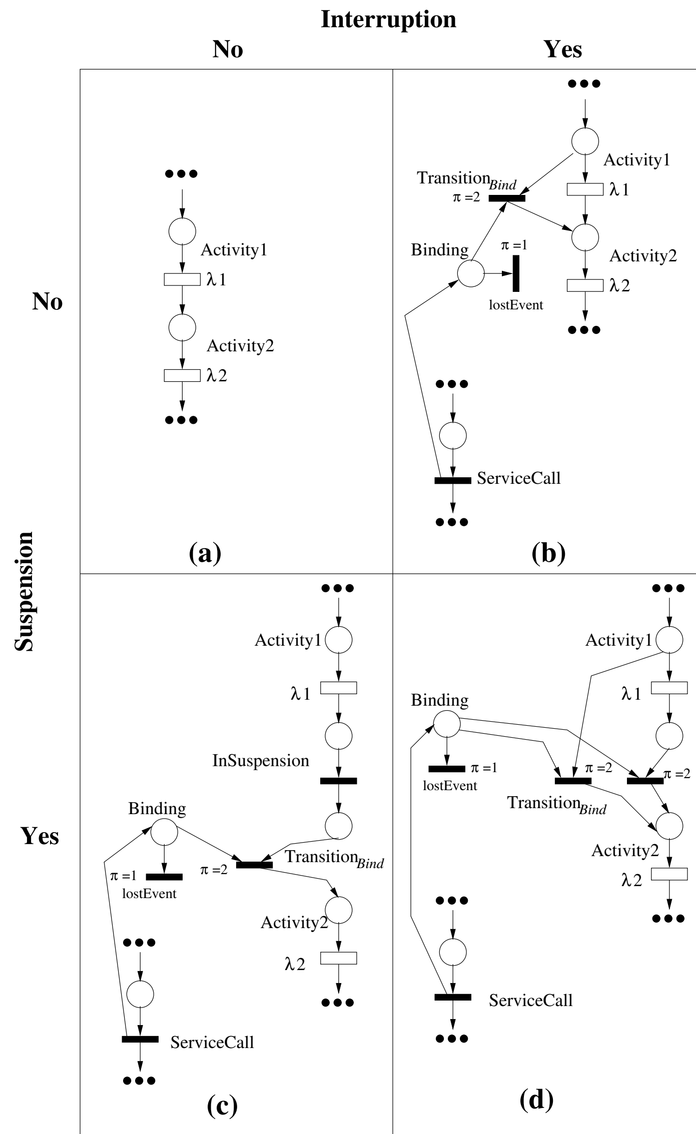
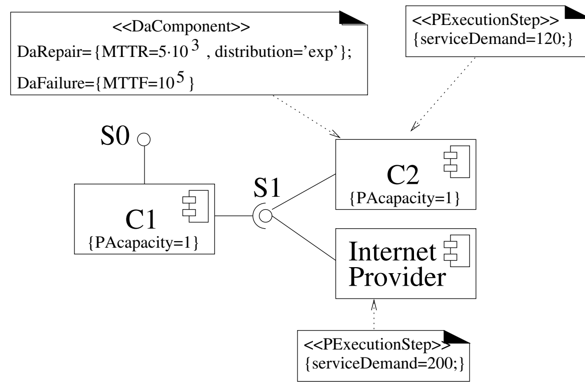
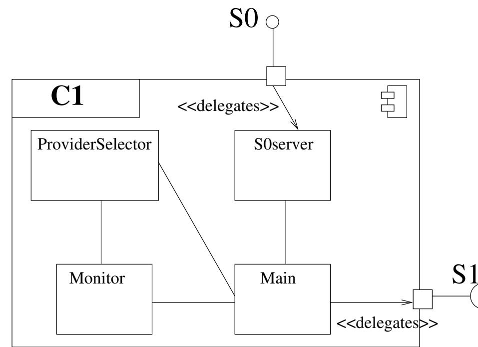


Figure 9.5: Petri net examples for combination of Interruption/Suspension/EventSending (transformation of Figure 9.2)



(a)



(b)

Figure 9.6: Components diagrams.

9.4.2 Reactive specification

Sequence diagram in Figure 9.7 offers a high-level view of the interactions in the system. When a Client asks for *S0*, the *S0server* class manages the request, then creating an instance of a *Main* object. The *Main* object cooperates with the system *Monitor* and *ProviderSelector* to effectively resolve the request. Figures 9.8, 9.9 and 9.10 respectively depict the UML state machines of the *Main*, *Monitor* and *ProviderSelector*. The behavior of *ProviderSelector* represents the AdaptationService of the system since it decides whether *S1* service calls will be requested to *C2* or to *InternetProvider*. classes.

The *Main* state machine will help to illustrate *suspension* and *event sending*. *Suspension* is accomplished by *CallingS1* state, when it is reached, the object will wait for the eventual arrival either of `restart` or `S1response`. Note that being the entry action execution immediate, the object is truly waiting for an event to react. Regarding *event sending* there are

several examples in this state machine, e.g., before entering in *Calling* state, *Main* sends the `start` event to the *Monitor*.

The other ability we identified for reactive behavior, i.e. *interruption*, is illustrated in the *Monitor* state machine. When *monitoring*, the `stop` event can interrupt the time-out execution to bring the *Monitor* to idle.

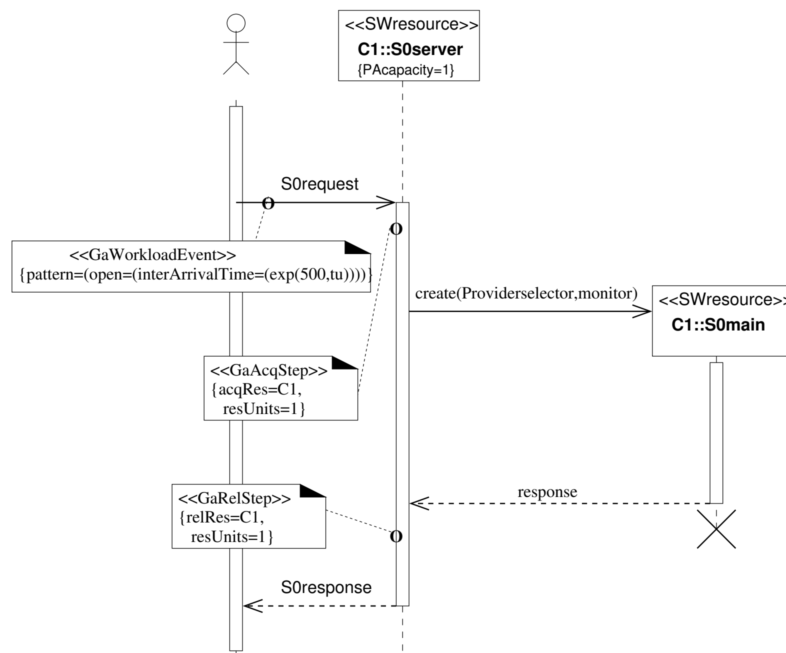


Figure 9.7: Sequence diagram representing the S0 requests.

9.4.3 Translation into D-KLAPER models

Figure 9.11 depicts the D-KLAPER corresponding to the sequence diagram in Figure 9.7, here it is important to note how the system workload is represented.

Figures 9.12, 9.13 and 9.14 depict D-KLAPER models for the UML state machines of the *Main*, *monitor* and *ProviderSelector* respectively. In the D-KLAPER model of the *Main* class, Figure 9.12, the call and the response to the external service *S1* (`CompBinded.S1` and `S1response` in the UML state machine) are translated as service calls. In the D-KLAPER model of the *ProviderSelector* class, Figure 9.14, the dashed part represents the necessary bindings for *S1* to be called, note that this is not yet specified in the UML model. However we assume this behavior is associated with the entry and exit actions in the *ProviderSelector* state machine states. Hence, note that this fact implies a D-KLAPER manual translation.

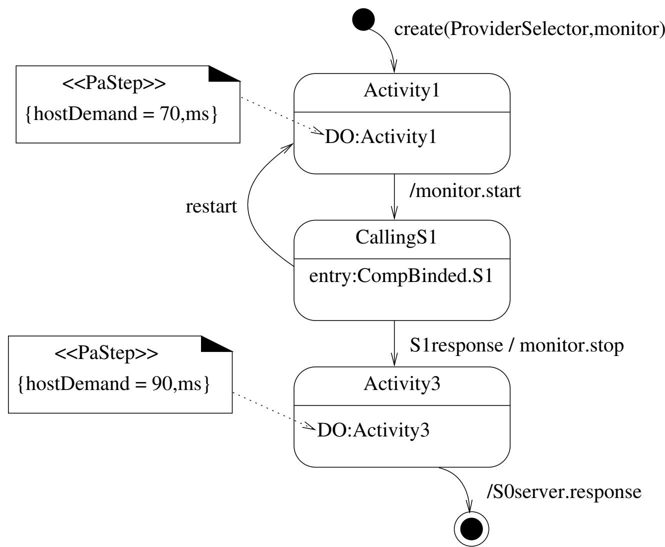


Figure 9.8: Main UML State machine

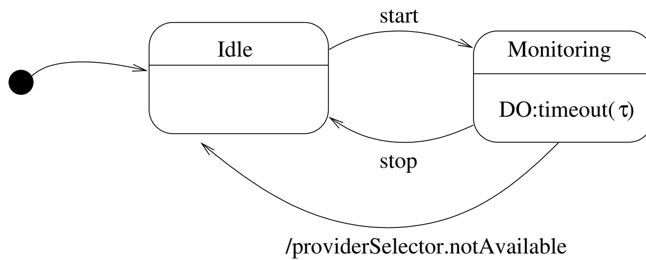


Figure 9.9: Monitor state machine

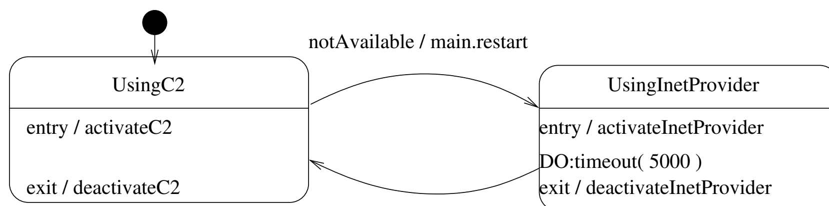


Figure 9.10: ProviderSelector state machine

Finally, the provider components D-KLAPER models appear in Figure 9.15. These two

models are obtained automatically translating the component diagram in Figure 9.6.

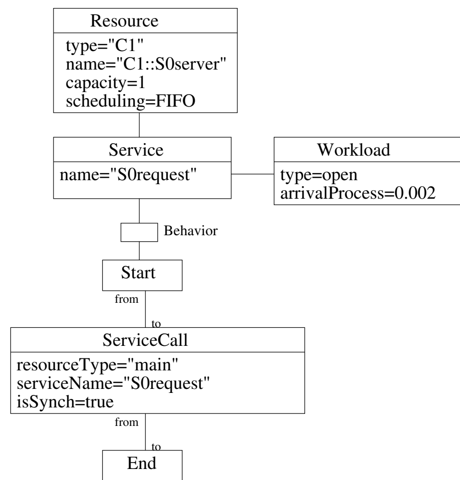


Figure 9.11: D-KLAPER model representing the sequence diagram in Fig. 9.7.

9.4.4 Translation into Petri nets and evaluation

Now, we proceed to evaluate the example, so to acquire knowledge and validate some non-functional properties. Concretely, the mean response time of *S0* and the mean monetary cost for an *S0* execution. Requirements of the system established that "the mean response time of *S0* has to be less than 700 time units (tu)" and "the mean cost for serving an *S0* request must not exceed two monetary units (mu)". A system restriction says that the *InternetProvider* offers each *S1* service call at a cost of 10 mu.

The D-KLAPER models in Figures 9.11 to 9.15 have been translated into a Deterministic and Stochastic Petri Net (DSPN) [AMC87], following the patterns in Figure 9.5 and ideas from [PPM10]. The obtained DSPN is then used to evaluate system performance and execution costs.

Variable τ in Figure 9.9 represents a threshold for the system to acknowledge *C2::S1* calls; upon expiration the monitor assumes that *C2* is no longer available. The higher τ is, the more *C2* will be used, then it may happen to the system to wait for *C2* while in fact it is unavailable. However, the lower τ is, the more the *InternetProvider* will be used, in this case the monitor may predict *C2* unavailability when it can be only performing an unusual slow service.

Figure 9.16 (a) depicts *S0* mean response time w.r.t. τ. The performance requirement is met from $\tau = 440$ (693.7tu) to τ=1640 (699.58tu) and a minimum is obtained around $\tau = 840$. Hence, timeouts lower than 440 confuse the system as explained in the previous paragraph, i.e., predicting erroneous *C2* unavailabilities. However, timeouts higher than 1640 lead the system to wait for *C2* even when it is dropped.

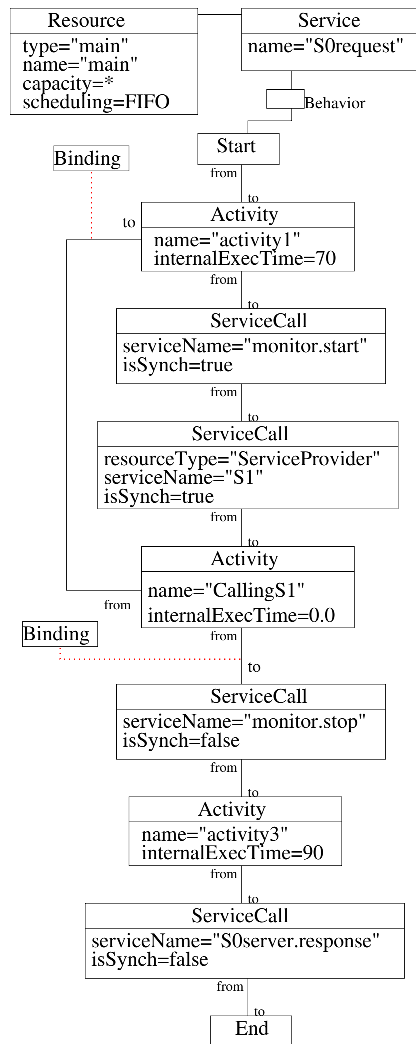


Figure 9.12: D-KLAPER model representing the main class.

Figure 9.16 (b) depicts the mean cost of executing *S0* w.r.t. τ. The function decreases because, as explained, *C2*, the free component, is more used for higher values of τ. The requirement is fulfilled for τ values upper than 520, since then, the mean cost is less than 2 mu.

Considering these two graphs, we observe that the non-functional requirements are met from τ=520 to τ=1640.

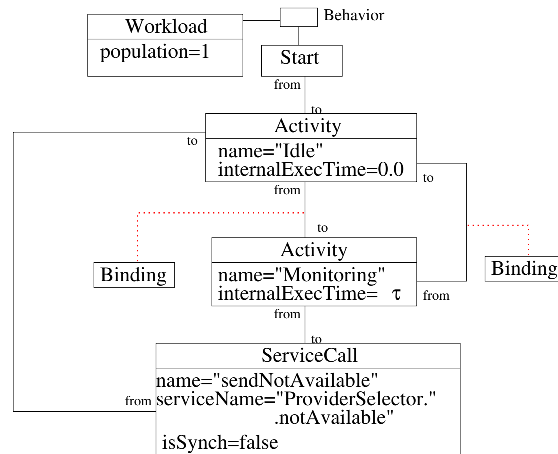


Figure 9.13: D-KLAPER model of the monitor state machine (in Figure 9.9).

9.5 Conclusion

In this chapter we have presented a Model-Driven approach whose goal is to support the QoS assessment of adaptable reactive systems. The approach follows the two-step model transformation process passing through the intermediate model D-KLAPER. We have extended the modeling power of D-KLAPER to capture the core features (from a performance/dependability viewpoint) of an adaptable reactive system model. This approach can be automated and implemented as a part of the KlaperSuite [CFD+11].

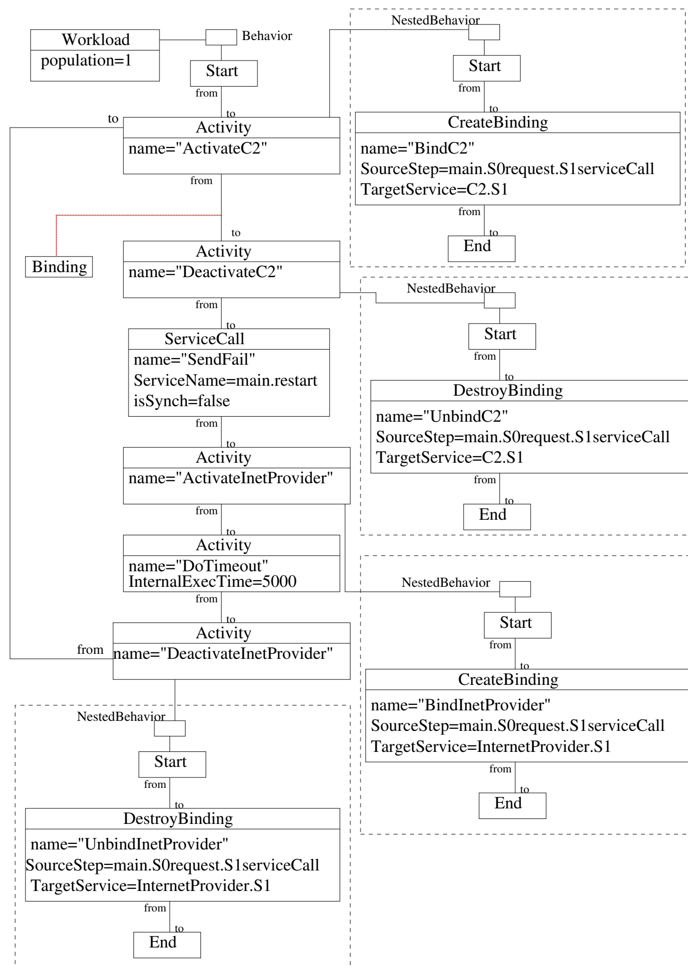


Figure 9.14: D-KLAPER model of the providerSelector statechart (in Figure 9.10).

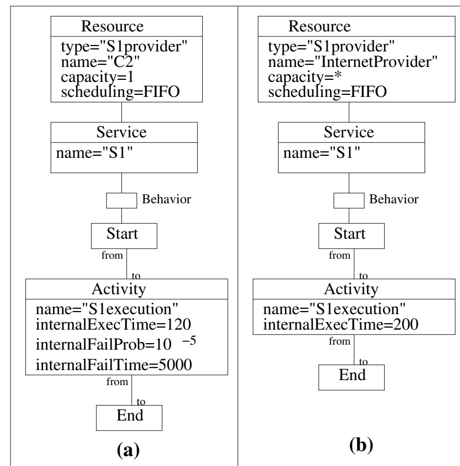
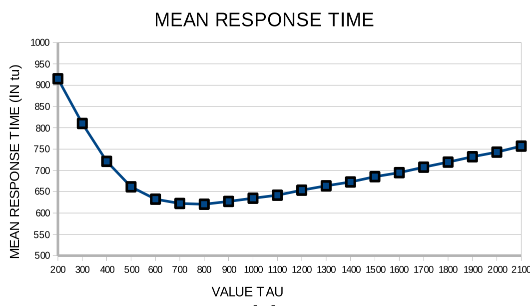


Figure 9.15: D-KLAPER models representing *S1* providers: (a) C2 (b) InternetServiceProvider



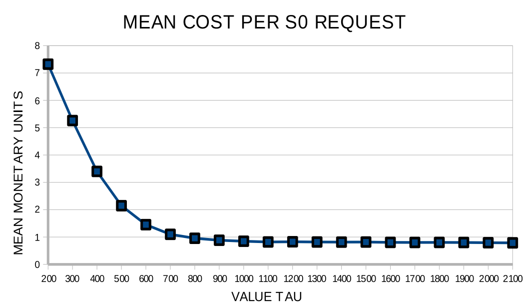


Figure 9.16: Results of the system evaluation

Chapter 10

Conclusion

In this thesis we have presented our research on the new and exciting paradigm of self-adaptive software.

Firstly, we believe it is very favorable the usage of an architetural approach for self-managed systems since it leverages greatly the rest of the research. Therefore, we tuned the concepts already proposed in a three-layer architecture for self-managed systems for the specific type of sofware that uses self-management capabilities to satisfy its extra functional requirements.

Using the architectural approach, we proposed solutions for the analysis and management of extra functional properties of software systems -such as performance and energy consumption. Moreover, we pursued reliable analysis results rather than specific ones coming from *ad-hoc* or newly implemented evaluation techniques. In order to meet that objective, the results we present are based on the utilization of formal methods such as stochastic Petri nets, hidden Markov models, or Markov-modulated Poisson processes. On the one hand, these formal methods provided us with accepted and verifiable analysis techniques whereas they allowed us to be confident of their results accuracy. That fact facilitated a part of our work because we could take advantage of the extensive research work already proposed for modeling and analyzing software systems using formalisms and trust its results. On the other hand, some modeling challenges had not been yet addressed; probably owing to its specific manifestation in self-adaptive systems -rather than to the broader software QoS analysis- and the novelty of these systems. One of these challenges is the modeling of the dynamic workload. We built-on theories for modeling workload already proposed and we presented a modeling approach that included in the formal model some important concepts that are specific to self-adaptive systems; for example, the transient times between workload changes. This research has produced [PPMB10, PPMM11a, PPMM11b, PPM11, PPMM12a, PPMM12b] published works.

For obtaining a formal model to analyze, we have followed Model-Driven Engineering principles. According to the recent proposal regarding the spliting of the transformation between software design model and analyzable model into two steps, we have used intermediate models already proposed for extra functional properties representation. We have presented

translation theories from two of these intermediate models to stochastic Petri nets. This research has produced [PPM10, PPMMG10] published works.

We have performed different experiments and examples using the proposed theories. We have presented the results of these experiments, which show that the enhancement of a software system with self-adaptive capabilities can be an opportunity for improving its extra functional properties.

The research performed in this thesis helps software developers to predict the expected behavior of the self-adaptive systems they construct. Furthermore, this thesis proposes theories for carrying out one of the most challenging tasks of self-adaptive software systems development, the adaptation plan generation considering tradeoffs between different extra functional properties of software.

Thinking of future work, there is plethora of challenges in sight. For example, research on fine-grained evaluation techniques for extra functional properties of self-managed software different from the ones we deeply worked in this thesis -e.g., security or safety. Furthermore, owing to the quick development of sofware engineering concepts for self-adaptive software, more challenges are daily arising. Some examples are the research on the prediction of self-adaptation capabilities in virtualized systems; or the predicition of software behavior when applying self-management theories to sofware services that are executed in the cloud. The analysis of the latter situation is delicated because it should be considered that the execution infrastructure offered by the cloud computing providers is in turn self-managed, and probably depending on the service behavior.

Relevant Publications Related to the Thesis

[PPCM13] Diego Perez-Palacin, Radu Calinescu, and José Merseguer. log2cloud: Log-based Prediction of Cost-Performance Trade-offs for Cloud Deployments. *To appear on SAC'13*.

[PPM10] Diego Perez-Palacin and José Merseguer. Performance evaluation of self-reconfigurable service-oriented software with stochastic Petri nets. *Electronic Notes in Theoretical Computer Science*, 261:181 – 201, 2010. Proceedings of PASM'09.

[PPM11] Diego Perez-Palacin and José Merseguer. Performance sensitive self-adaptive service-oriented software using hidden Markov models. In *Proceedings of WOSP/SIPEW '11*, pages 201–206, 2011.

[PPMB10] Diego Perez-Palacin, José Merseguer, and Simona Bernardi. Performance aware open-world software in a 3-layer architecture. In *Proceedings of WOSP/SIPEW '10*, pages 49–56, 2010.

[PPMB12] Diego Perez-Palacin, José Merseguer, and Simona Bernardi. Performance aware self-managed software: evaluation using Petri nets. *Submitted to a journal*.

[PPMM11a] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. Enhancing a QoS-based self-adaptive framework with energy management capabilities. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 165–170, New York, NY, USA, 2011. ACM.

[PPMM11b] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. Software architecture adaptability metrics for QoS-based self-adaptation. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 171–176, New York, NY, USA, 2011. ACM.

[PPMM12a] Diego Perez-Palacin, José Merseguer, and Raffaela Mirandola. Analysis of bursty workload-aware self-adaptive systems. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 75–84, New York, NY, USA, 2012. ACM.

[PPMM12b] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. QoS and energy management with Petri nets: A self-adaptive framework. *Journal of Systems and Software*, 85(12):2796 – 2811, 2012.

[PPMMG10] Diego Perez-Palacin, Raffaela Mirandola, José Merseguer, and Vincenzo Grassi. QoS-based model driven assessment of adaptive reactive systems. In *ICST Workshops*, pages 299–308. IEEE Computer Society, 2010.

Appendixes

Appendix A

Generalized Stochastic Petri Nets

A PN system is a tuple PN, where P and T are the sets of *places* and *transitions*, $\mathbf{Pre}$ and $\mathbf{Post}$ are the $|P|\times|T|$ sized, natural valued, pre- and post- incidence matrices. For instance, $\mathbf{Post}[p,t] = w$ means that there is an *arc* from t to p with *multiplicity* w. When all weights are one, the PN is *ordinary*. $\mathbf{C} = \mathbf{Post} - \mathbf{Pre}$ is the *incidence matrix* of the net. For pre- and postsets we use the conventional dot notation, e.g., ${}^{\bullet}t = \{p \in P : \mathbf{Pre}[p,t] \geq 1\}$, that can be extended to sets of nodes. If $\mathcal{N}'$ is the subnet of $\mathcal{N}$, defined by $P' \subseteq P$ and $T' \subseteq T$, then $\mathbf{Pre}' = \mathbf{Pre}[P',T']$, $\mathbf{Post}' = \mathbf{Post}[P',T']$ and $\mathbf{M_0'} = \mathbf{M_0}[P']$. Subnets defined by a subset of places (transitions), with all their adjacent transitions (places), are called P- (T-) subnets.

A *marking* $\mathbf{M}$ is a $|P|$ sized, natural valued, vector and $\mathbf{M_0}$ is the *initial marking* vector. A transition is *enabled* in $\mathbf{M}$ iff $\mathbf{M} \geq \mathbf{Pre}[P,t]$; its *firing*, denoted by $\mathbf{M} \xrightarrow{t} \mathbf{M'}$, yields a new marking $\mathbf{M'} = \mathbf{M} + \mathbf{C}[P,t]$. The set of all reachable markings is denoted as $RS(\mathcal{N}, \mathbf{M_0})$. An *occurrence sequence* from $\mathbf{M}$ is a sequence of transitions $\sigma = t_1 \ldots t_k \ldots$ such that $\mathbf{M} \xrightarrow{t_1} \mathbf{M_1} \ldots M_{k-1} \xrightarrow{t_k} \ldots$. Given σ such that $\mathbf{M} \xrightarrow{\sigma} \mathbf{M'}$, and denoting by σ the $|T|$ sized firing count vector of σ, then $\mathbf{M'} = \mathbf{M} + \mathbf{C} \cdot \sigma$ is known as the *state equation* of $\mathcal{N}$.

A GSPN is a tuple $\mathcal{G} = (\mathcal{N}, \mathbf{\Pi}, \bar{\mathbf{S}}, \mathbf{r})$, where $\mathcal{N}$ is a PN system and the set of transitions T is partitioned in two subsets T_t and T_i of timed and immediate transitions, respectively. $\mathbf{\Pi}$ is a natural valued, $|\,T\,|$ sized, vector that specifies a priority level of each transition. Timed transitions have zero priority, immediate transitions have priority greater than zero. A transition $t \in T$, enabled in marking $\mathbf{M}$, can fire if no transition $t' \in T : \mathbf{\Pi}[t'] > \mathbf{\Pi}[t]$ is enabled in $\mathbf{M}$. Timed transition firing delays are random variables, distributed according to negative exponential probability distribution functions. Immediate transitions fire instead in zero time. $\bar{\mathbf{S}}$ is a non negative real valued, $|\,T_t\,|$ sized, vector of the mean transition firing times. The positive real valued vector $\mathbf{r}$ is $|T_i|$ sized, and specifies the weights of immediate transitions for probabilistic conflict resolution.

Appendix B

GSPN Composition

B.1 MLGSPN Composition

The definition of the composition of multi-labeled generalized stochastic Petri nets (MLGSPN) is an extension of the definitions in [DF96, BDH01, Ber03].

In [AMBC+95] a GSPN is defined as a 8-tuple $\mathcal{S} = (P, T, \Pi, I, O, H, W, M^0)$, where P is the set of places, T is the set of transitions (immediate and timed), $P \cap T = \emptyset$; $\Pi : T \rightarrow \mathbb{N}$ is the priority function that assigns a priority level to each transition. $I, O, H : T \rightarrow 2^P$ are the input,output, inhibition functions, respectively, that map transitions onto the power set of P; $W : T \rightarrow \mathbb{R}$ is the weight function that assigns rates of timed transitions and weights to immediate transitions. $M^0 : P \rightarrow \mathbb{N}$ is the initial marking of the net.

In [Ber03] a multi-labeled GSPN (MLGSPN) is defined as a triplet $\mathcal{M}LS = (\mathcal{S}, \Psi, \Lambda)$, where $\mathcal{S}$ is a GSPN model, as defined above, $\Psi : P \rightarrow 2^{L^P}$ is the labeling function that assigns to a place a set of labels belonging to 2^{L^P}, where 2^{L^P} is the power-set of the label of the places L^P, and $\Lambda : T \rightarrow 2^{L^T}$ is the labeling function that assigns to a transition a set of labels belonging to 2^{L^T}.

Given two MLGSPN models $\mathcal{M}LS_1 = (\mathcal{S}_1, \Psi_1, \Lambda_1)$ and $\mathcal{M}LS_2 = (\mathcal{S}_2, \Psi_2, \Lambda_2)$, we formally define the MLGSPN composition over the set of labels L_T and L_P as:

$$\mathcal{M}LS = \mathcal{M}LS_1 \underset{L_T, L_P}{||} \mathcal{M}LS_2$$

The resulting transitions and places of $\mathcal{M}LS$ are the sets:
$T = T_1 \backslash T_1^{E_T} \cup T_2 \backslash T_2^{E_T} \cup T_{NEW}$
$P = P_1 \backslash P_1^{E_P} \cup P_2 \backslash P_2^{E_P} \cup P_{NEW}$
where:

- $T_2^{E_T}$ ($P_2^{E_P}$) as the set of all transitions (places) in $\mathcal{M}LS_2$ which are labeled by any label in the set E_T (E_P).

- $E_T = L_T \cap \Lambda_1(T_1) \cap \Lambda_2(T_2)$

- $E_P = L_P \cap \Psi_1(P_1) \cap \Psi_2(P_2)$
- $T_i \backslash T_i^{E_T}$ ($P_i \backslash P_i^{E_P}$) means the transitions (places) in $\mathcal{S}_i$ that will not be composed.
- $T_{NEW} = \bigcup_{t_i \in T_1} (t_i \times (\otimes_{l \in \{\Lambda_1(t_i) \cap E_T\}} \{T_2^l\}))$
- $P_{NEW} = \bigcup_{p_i \in P_1} (p_i \times (\otimes_{l \in \{\Psi_1(p_i) \cap E_P\}} \{P_2^l\}))$
- $\otimes$ is the Cartesian product.
- T_2^l (P_2^l) is defined as the set of transitions (places) in $\mathcal{M}LS_2$ being one of their labels l. Note that $\bigcup_{l \in E_T} T_1^l = T_1^{E_T}$.

Let:

- $\sigma_T : \bigcup_{L \subseteq E_T} \{T_1^L \times T_2^L \times ... \times T_2^L\} \rightarrow T_{NEW}$ be a bijection that assigns a new transition to each set of transitions $(t_1, t_2, ..., t_n) \in T_1 \times T_2 \times ... \times T_2$ where $L \equiv (\Lambda_1(t_1) \cap E_T)$, for all $j, k = \{2...n\}\{(j \neq k) \Rightarrow (t_j \neq t_k)\}$ and for all $i = \{2...n\}(L \cap \Lambda_2(t_i)) \neq \emptyset$.
- $\sigma_P : \bigcup_{L \subseteq E_P} \{P_1^L \times P_2^L \times ... \times P_2^L\} \rightarrow P_{NEW}$ be a bijection that assigns a new place to each set of places $(p_1, p_2, ..., p_n) \in P_1 \times P_2 \times ... \times P_2$ where $L \equiv (\Psi_1(p_1) \cap E_P)$, for all $j, k = \{2...n\}\{(j \neq k) \Rightarrow (p_j \neq p_k)\}$ and for all $i = \{2...n\}(L \cap \Psi_2(p_i)) \neq \emptyset$.

Function $F \in \{I(), O(), H()\}$ of $\mathcal{M}LS$ is:
(a) **case** $t \in T_1 \backslash T_1^{E_T}$:

$$F(t) = F_1(t) \backslash (F_1(t) \cap P_1^{E_P})$$
$$\bigcup$$
$$\bigcup_{p \in P_{NEW}} (p \mid \sigma_P(p_1, p_2, ..., p_j) \equiv p \wedge (F_1(t) \cap p_1) \neq \emptyset)$$

(b) **case** $t \in T_2 \backslash T_2^{E_T}$:

$$F(t) = F_2(t) \backslash (F_2(t) \cap P_2^{E_P})$$
$$\bigcup$$
$$\left(\begin{array}{c} \bigcup_{p \in P_{NEW}} (p \mid \sigma_P(p_1, p_2, ..., p_j) \equiv p \\ \wedge \\ \exists i \in \{2, ..., j\}(F_2(t) \cap p_i) \neq \emptyset) \end{array} \right)$$

(c) **case** $t \in T_{NEW} \wedge t \equiv \sigma_T(t_1, t_2, ..., t_j)$:

$$
\begin{array}{c}
F(t) = F_1(t_1)\backslash(F_1(t_1) \cap P_1^{E_P}) \\
\bigcup \\
\bigcup_j^{i=2}(F_2(t_i)\backslash(F_2(t_i) \cap P_2^{E_P})) \\
\bigcup \\
\bigcup_{p \in P_{NEW}}(p \mid \sigma_P(p_1, p_2, ..., p_j) \equiv p \wedge (F_1(t_1) \cap p_1) \neq \emptyset) \\
\bigcup \\
\left(\begin{array}{c}
\bigcup_{k=2..j}(\bigcup_{p \in P_{NEW}}(p \mid \sigma(p_1, p_2, ..., p_j) \equiv p \\
\wedge \\
\exists i \in \{2, ..., j\}(F_2(t_k) \cap p_i) \neq \emptyset))
\end{array} \right)
\end{array}
$$

Function $W()$ of $\mathcal{M}LS$ is:

$$
W(t) = \begin{cases}
W_1(t) & \text{if} \quad t \in T_1\backslash T_1^{E_T} \\
W_2(t) & \text{if} \quad t \in T_2\backslash T_2^{E_T} \\
min(W_1(t_1), min_{i=2..j}(W_2(t_i))) & \text{if} \quad t \in T_{NEW} \wedge \\
& t \equiv \sigma_T(t_1, t_2, ..., t_j)
\end{cases}
$$

Function Π of $\mathcal{M}LS$ is:

$$
\Pi(t) = \begin{cases}
\Pi_1(t) & \text{if} \quad t \in T_1\backslash T_1^{E_T} \\
\Pi_2(t) & \text{if} \quad t \in T_2\backslash T_2^{E_T} \\
min(\Pi_1(t_1), min_{i=2..j}(\Pi_2(t_i))) & \text{if} \quad t \in T_{NEW} \wedge \\
& t \equiv \sigma_T(t_1, t_2, ..., t_j)
\end{cases}
$$

Function M^0 of $\mathcal{M}LS$ is:

$$
M^0(p) = \begin{cases}
M_1^0(p) & \text{if} \quad p \in P_1\backslash P_1^{E_P} \\
M_2^0(p) & \text{if} \quad p \in P_2\backslash P_2^{E_P} \\
M_1^0(p_1) + \Sigma_{i=2}^{j} M_2^0(p_i) & \text{if} \quad p \in P_{NEW} \wedge \\
& p \equiv \sigma_P(p_1, p_2, ...p_j)
\end{cases}
$$

Labeling functions $\Psi()$ and $\Lambda()$ of $\mathcal{M}LS$ are:

$$
\Psi(x) = \begin{cases}
\Psi_1(x) & \text{if} \quad x \in P_1\backslash P_1^{E_P} \\
\Psi_2(x) & \text{if} \quad x \in P_2\backslash P_2^{E_P} \\
\Psi_1(p_1) \cup (\bigcup_{i=2..j} \Psi_2(p_i)) & \text{if} \quad x \in P_{NEW} \wedge \\
& x \equiv \sigma_P(p_1, p_2, ..., p_j)
\end{cases}
$$

$$\Lambda(x) = \begin{cases} \Lambda_1(x) & \text{if} \quad x \in T_1 \backslash T_1^{E_T} \\ \Lambda_2(x) & \text{if} \quad x \in T_2 \backslash T_2^{E_T} \\ \Lambda_1(t_1) \cup (\bigcup_{i=2..j} \Lambda_2(t_i)) & \text{if} \quad x \in T_{NEW} \wedge \\ & \quad x \equiv \sigma_T(t_1, t_2, ..., t_j)) \end{cases}$$

An example of composition is shown in Figure B.1. Figure 8.3(c) in Chapter 8 is an example of the resulting priority function Π when (a) and (b) are composed using L_T. The `DB_acq` transition in (a) is assigned an infinite priority. Therefore, when (a) is composed with (b), the resulting transition `DB_acq` has the lower priority, i.e. 2. The composition operator does not satisfy the commutative property.

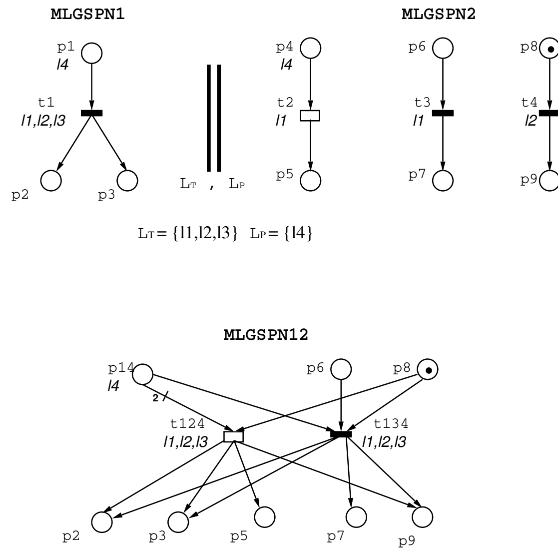


Figure B.1: MLGSPN composition

Bibliography

[AAA+06] Bruno D. Abrahao, Virgilio Almeida, Jussara M. Almeida, Alex Zhang, Dirk Beyer, and Fereydoon Safai. Self-adaptive SLA-driven capacity management for internet services. pages 557 –568, apr. 2006.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.

[AGM08] Danilo Ardagna, Carlo Ghezzi, and Raffaela Mirandola. Rethinking the use of models in software architecture. In *QoSA*, volume 5281 of *Lecture Notes in Computer Science*, pages 1–27, 2008.

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36 – 41, sept.-oct. 2003.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[AMBC+95] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley Series in Parallel Computing - Chichester, 1995.

[AMC87] Marco Ajmone Marsan and Giovanni Chiola. On Petri nets with deterministic and exponentially distributed firing times. In *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets*, pages 132–145, London, UK, 1987. Springer-Verlag.

[Arm00] Phillip G. Armour. The five orders of ignorance. *Commun. ACM*, 43(10):17–20, October 2000.

[Bal98] Gianfranco Balbo. Non–exponential stochastic Petri nets. In *Performance Models for Discrete Event Systems with Synchronizations: Formalisms and Analysis Techniques*, pages 345–386. KRONOS, Zaragoza, Spain, 1998.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Jürgen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS*, pages 28–33. ACM, 2004.

[BCK05] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Boston ; Munich [u.a.], 2005.

[BCvH+03] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri net markup language: Concepts, technology, and tools. In *ICATPN*, volume 2679 of *LNCS*, pages 483–505, 2003.

[BDH01] Simona Bernardi, Susanna Donatelli, and András Horváth. Implementing compositionality for stochastic Petri nets. *STTT*, 3(4):417–430, 2001.

[BDIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. on Software Engineering*, 30(5):295–310, May 2004.

[Ber03] Simona Bernardi. *Building Stochastic Petri Net models for the verification of complex software systems*. PhD thesis, Dipartimento di Informatica, Università di Torino, April 2003.

[BGG04] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 193–202, New York, NY, USA, 2004. ACM.

[BJR99] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language*. Addison Wesley, 1999.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1), 2009.

[BLM10] Luciano Bertini, Julius C.B. Leite, and Daniel Mossé. Power optimization for dynamic configuration in heterogeneous web server clusters. *Journal of Systems and Software*, 83(4):585 – 598, 2010.

[BMP11] Simona Bernardi, José Merseguer, and Dorina C. Petriu. A dependability profile within marte. *Software and Systems Modeling*, 10(3):313–336, July 2011.

[BNG06] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

[BR04] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *Computer*, 37(11):68 – 76, 2004.

[Buc03] Peter Buchholz. An em-algorithm for map fitting from real traffic data. In Peter Kemper and William H. Sanders, editors, *Computer Performance Evaluation / TOOLS*, volume 2794 of *Lecture Notes in Computer Science*, pages 218–236. Springer, 2003.

[CCG+09] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 131–140, 2009.

[CD01] Jeff Chase and Ron Doyle. Balance of power: Energy management for server clusters. In *HotOS'01*, May 2001.

[CdLG+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.

[CDPEV08] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A framework for QoS-aware binding and re-binding of composite web services. *J. Syst. Softw.*, 81(10):1754–1769, 2008.

[CDQ+05] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. *SIGMETRICS Perform. Eval. Rev.*, 33(1):303–314, 2005.

[CFD+11] Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaela Mirandola, and Vincenzo Grassi. Klapersuite: an integrated model-driven environment for reliability and performance analysis of component-based systems. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11, pages 99–114, Berlin, Heidelberg, 2011. Springer-Verlag.

[CGK+11] Radu Calinescu, Lars Grunske, Marta Z. Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Trans. Software Eng.*, 37(3):387–409, 2011.

[CJH+11] Shuyi Chen, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and William H. Sanders. Using cpu gradients for performance-aware energy conservation in multitier systems. *Sustainable Computing: Informatics and Systems*, 2011.

[CKK01] Paul C. Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. SEI Series in Software Engineering. Addison-Wesley, 2001.

[CLG+09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns,

and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. pages 1–26, 2009.

[CMCS12] Giuliano Casale, Ningfang Mi, Ludmila Cherkasova, and Evgenia Smirni. Dealing with burstiness in multi-tier applications: Models and their parameterization. *IEEE Trans. Software Eng. To appear*, 2012.

[CMI07] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software*, 80(4):455–473, 2007.

[CMI11] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[CNYM99] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering (The Kluwer International Series in Software Engineering Volume 5)*. Springer, 1st edition, October 1999.

[CPR07] Vittorio Cortellessa, Pierluigi Pierini, and Daniele Rossi. Integrating software models and platform models for performance analysis. *IEEE Transactions on Software Engineering*, 33:385–401, 2007.

[CS93] Javier Campos and Manuel Silva. Embedded product-form queueing networks and the improvement of performance bounds for petri net systems. *Perform. Eval.*, 18(1):3–19, 1993.

[CS01] Lawrence Chung and Nary Subramanian. Process-oriented metrics for software architecture adaptability. In *RE*, pages 310–311. IEEE Computer Society, 2001.

[CSM] CSM to GSPN Translator. http://webdiis.unizar.es/~jmerse/csm2pn.html.

[CVP+08] Ítalo Cunha, Itamar Viana, João Palotti, Jussara M. Almeida, and Virgilio Almeida. Analyzing security and energy tradeoffs in autonomic capacity management. pages 302 –309, apr. 2008.

[CZS10] Giuliano Casale, Eddy Z. Zhang, and Evgenia Smirni. KPC-Toolbox: Best recipes for automatic trace fitting using Markovian Arrival Processes. *Perform. Eval.*, 67:873–896, September 2010.

[DAR] DARPA. Self adaptive software. DARPA, BAA 98-12, Proposer Information Pamphlet, December.

[DDF+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1:223–259, December 2006.

[DF96] Susanna Donatelli and Giuliana Franceschinis. PSR Methodology: integrating hardware and software models. In *ICATPN*, volume 1091 of *LNCS*, pages 133–152, 1996.

[DKL+08] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan. Autonomic multi-agent management of power and performance in data centers. In *AAMAS '08*, pages 107–114, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[DN02] Liliana Dobrica and Eila Niemelä;. A survey on software architecture analysis methods. *IEEE Trans. on Software Engineering*, 28(7):638–653, Jul 2002.

[DSP11] Salvatore Distefano, Marco Scarpa, and Antonio Puliafito. From UML to Petri nets: The PCM-based methodology. *IEEE Transactions on Software Engineering*, 37:65–79, 2011.

[Edd04] Sean R. Eddy. What is a hidden Markov model? *Nature Biotechnology*, 22(10):1315–1316, October 2004.

[EKR03] E. N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. PACS'02, pages 179–197, 2003.

[FMH93] Wolfgang Fischer and Kathleen Meier-Hellstern. The Markov-modulated Poisson process (MMPP) cookbook. *Perform. Eval.*, 18:149–171, September 1993.

[Fra10] Estíbaliz Fraca. Implementación de un generador de estrategias de autoconfiguración para la mejora de prestaciones de software autoadaptativo, March 2010. Master Sc Thesis. In spanish.

[GBMP97] Erann Gat, R. Peter Bonnasso, Robin Murphy, and Aaai Press. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GMM06] Elena Gómez-Martínez and José Merseguer. Argospe: Model-based software performance engineering. volume 4024, pages 401–410. Springer-Verlag, Springer-Verlag, 2006.

[GMMT10] Carlo Ghezzi, Alfredo Motta, Valerio Panzica La Manna, and Giordano Tamburrelli. QoS driven dynamic binding in-the-many. In *QoSA*, pages 68–83, 2010.

[GMR09] Vincenzo Grassi, Raffaela Mirandola, and Enrico Randazzo. Model-Driven assessment of QoS-Aware Self-Adaptation. In *Software Engineering for Self-Adaptive Systems*, pages 201–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[GMS07a] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, 2007.

[GMS07b] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 103–114, New York, NY, USA, 2007. ACM.

[Gre] The GreatSPN tool. `http://www.di.unito.it/~greatspn`.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM.

[GT09] Carlo Ghezzi and Giordano Tamburrelli. Predicting performance properties for open systems with kami. In *QoSA*, pages 70–85, 2009.

[Gus91] Riccardo Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *Selected Areas in Communications, IEEE Journal on*, 9(2):203 –211, feb 1991.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Hav95] Boudewijn R. Haverkort. Matrix-geometric solution of infinite stochastic petri nets. In *In Proceedings of the First International Computer Performance and Dependability Symposium*, pages 72–81. IEEE Computer Society Press, 1995.

[HBK11] Nikolaus Huber, Fabian Brosig, and Samuel Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 90–99, New York, NY, USA, 2011. ACM.

[HBR^{+}10] Jens Happe, Steffen Becker, Christoph Rathfelder, Holger Friedrich, and Ralf H. Reussner. Parametric performance completions for model-driven performance prediction. *Perform. Eval.*, 67:694–716, August 2010.

[HL86] Harry Heffes and David M. Lucantoni. A markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance. *Selected Areas in Communications, IEEE Journal on*, 4(6):856 – 868, sep 1986.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In *In K. R. Apt, editor, Logics and Models of Concurrent Systems, volume F-13 of NATO ASI Series*, pages 477–498, 1985.

[HT02] András Horváth and Miklós Telek. Markovian modeling of real data traffic: Heuristic phase type and map fitting of heavy tailed and fractal like samples. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, pages 405–434. Springer-Verlag, 2002.

[IDC] IDC. `http://www.idc.com`.

[IFMW08] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *SEAMS*, 2008.

[JAVa] Java technology. `http://www.sun.com/java/`.

[javb] java.lang.Math specification. http://docs.oracle.com/javase/6/docs/api/java-/lang/Math.html .

[JHJ+10] Gueyoung Jung, Matti A. Hiltunen, Kaustubh R. Joshi, Richard D. Schlichting, and Calton Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 62–73, 2010.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KKH+09] Dara Kusic, Jeffrey Kephart, James Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12:1–15, 2009.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, 2007.

[KM09] Jeff Kramer and Jeff Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24(2):183–188, March 2009.

[KRG+10] Elsy Kaddoum, Claudia Raibulet, Jean-Pierre Georgé, Gauthier Picard, and Marie-Pierre Gleizes. Criteria for the evaluation of self-* systems. In *SEAMS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 29–38, New York, NY, USA, 2010. ACM.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Addison Wesley Object Technology Series, 2003.

[Lad99] Robert Laddaga. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14:26–29, 1999.

[Lad00] Robert Laddaga. Active software. In *IWSAS' 2000: Proceedings of the first international workshop on Self-adaptive software*, pages 11–26, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[LGMC04] J.P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic Petri nets: Application to software performance engineering. In *Proceedings of WOSP'04*, pages 25–36, Redwood City, California, USA, January 2004. ACM.

[LR04] Robert Laddaga and Paul Robertson. Self adaptive software: A position paper. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, 2004.

[MA01] D.A. Menascé and V.A. F. Almeida. *Capacity Planning for Web Services: metrics, models, and methods.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[MCCS08] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Burstiness in multi-tier applications: symptoms, causes, and new models. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 265–286, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[MCL+01] John Mylopoulos, Lawrence Chung, Stephen Liao, Huaiqing Wang, and Eric Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18:92–96, 2001.

[MD07] D.A. Menasce and V. Dubey. Utility-based QoS brokering in service oriented architectures. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 422 –430, july 2007.

[MM06] Antinisca Di Marco and Raffaela Mirandola. Model transformation in software performance engineering. In *QoSA*, volume 4214 of *Lecture Notes in Computer Science*, pages 95–110, 2006.

[MRG07] Daniel A. Menascé, Honglei Ruan, and Hassan Gomaa. Qos management in service-oriented architectures. *Perform. Eval.*, 64(7-8):646–663, August 2007.

[MZR+07] Ningfang Mi, Qi Zhang, Alma Riska, Evgenia Smirni, and Erik Riedel. Performance impacts of autocorrelated flows in multi-tiered systems. *Perform. Eval.*, 64:1082–1101, October 2007.

[Obj02] Object Management Group. UML 2.0 superstructure specification, 2002.

[Obj05] Object Management Group, `http://www.promarte.org`. *A UML Profile for MARTE.*, 2005.

[OD09] Hiroyuki Okamura and Tadashi Dohi. Faster maximum likelihood estimation algorithms for markovian arrival processes. QEST '09, pages 73–82. IEEE Computer Society, 2009.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[PPM10] Diego Perez-Palacin and José Merseguer. Performance evaluation of self-reconfigurable service-oriented software with stochastic petri nets. *Electronic Notes in Theoretical Computer Science*, 261:181 – 201, 2010. Proceedings of PASM'09.

[PPM11] Diego Perez-Palacin and José Merseguer. Performance sensitive self-adaptive service-oriented software using hidden Markov models. In *Proceedings of WOSP/SIPEW '11*, pages 201–206, 2011.

[PPMB10] Diego Perez-Palacin, José Merseguer, and Simona Bernardi. Performance aware open-world software in a 3-layer architecture. In *Proceedings of WOSP/SIPEW '10*, pages 49–56, 2010.

[PPMM11a] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. Enhancing a qos-based self-adaptive framework with energy management capabilities. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 165–170, New York, NY, USA, 2011. ACM.

[PPMM11b] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. Software architecture adaptability metrics for qos-based self-adaptation. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 171–176, New York, NY, USA, 2011. ACM.

[PPMM12a] Diego Perez-Palacin, José Merseguer, and Raffaela Mirandola. Analysis of bursty workload-aware self-adaptive systems. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 75–84, New York, NY, USA, 2012. ACM.

[PPMM12b] Diego Perez-Palacin, Raffaela Mirandola, and Jos Merseguer. Qos and energy management with petri nets: A self-adaptive framework. *Journal of Systems and Software*, 85(12):2796 – 2811, 2012.

[PPMMG10] Diego Perez-Palacin, Raffaela Mirandola, José Merseguer, and Vincenzo Grassi. Qos-based model driven assessment of adaptive reactive systems. In *ICST Workshops*, pages 299–308. IEEE Computer Society, 2010.

[PSL03] Chintan Patel, Kaustubh Supekar, and Yugyung Lee. A qos oriented framework for adaptive management of web service based workflows. In *In Proceeding of Database and Expert Systems 2003 Conference*, pages 826–835. Springer, 2003.

[PT07] Andriy Panchenko and Axel Thümmler. Efficient phase-type fitting with aggregated traffic traces. *Perform. Eval.*, 64:629–645, August 2007.

[PUMa] Core Scenario Model schema. `http://www.sce.carleton.ca/rads/puma/csm-metamodel/CSM.xsd`.

[PUMb] PUMA project web page. `http://sce.carleton.ca/rads/puma`.

[PW07] Dorin Petriu and Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and Systems Modeling (SoSyM)*, 6(2):163–184, June 2007.

[Rab89] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Ran10] Parthasarathy Ranganathan. Recipe for efficiency: principles of power-aware computing. *Commun. ACM*, 53:60–67, April 2010.

[RM09] Claudia Raibulet and Laura Masciadri. Evaluation of dynamic adaptivity through metrics: an achievable target? In *WICSA/ECSA*, pages 341–344, 2009.

[RWvM10] Philipp Reinecke, Katinka Wolter, and Aad P. A. van Moorsel. Evaluating the adaptivity of computing systems. *Perform. Eval.*, 67(8):676–693, 2010.

[Ryd96] Tobias Rydén. An EM algorithm for estimation in Markov-modulated Poisson processes. *Comput. Stat. Data Anal.*, 21:431–447, April 1996.

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17, 2001.

[SC01] Nary Subramanian and Lawrence Chung. Metrics for software adaptability. In *Proc. Software Quality Management*, pages 95–108, 2001.

[Sch06a] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25 – 31, feb. 2006.

[Sch06b] Philip Schrodt. Forecasting conflict in the balkans using hidden markov models. In Robert Trappl, Melvin F. Shakun, Tung Bui, Guy Olivier Faure, Gregory Kersten, D. Marc Kilgour, and Peyman Faratin, editors, *Programming for Peace*, volume 2 of *Advances in Group Decision and Negotiation*, pages 161–184. Springer Netherlands, 2006. 10.1007/1-4020-4390-2_8.

[SM05] Monchai Sopitkamol and Daniel A. Menascé. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 53–64, New York, NY, USA, 2005. ACM.

[Smi90] C. U. Smith. *Performance Engineering of Software Systems*. The Sei Series in Software Engineering. Addison–Wesley, 1990.

[ST09] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[SW02a] Connie U. Smith and Lloyd G. Williams. *Performance and Scalability of Distributed Software Architectures: an SPE Approach*. Addison Wesley, 2002.

[SW02b] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[SWHB06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 239–252, Berkeley, CA, USA, 2006. USENIX Association.

[TC11] Teik-Toe Teoh and Siu-Yeung Cho. Human emotional states modeling by hidden markov model. In *Natural Computation (ICNC), 2011 Seventh International Conference on*, volume 2, pages 908 –912, july 2011.

[TGEM10] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *ASE '10*, pages 467–476, New York, NY, USA, 2010. ACM.

[USC+08] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3:1:1–1:39, March 2008.

[WC03] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, 2003.

[Wor98] World Cup 1998 Access logs. `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`. 1998.

[wos10] WOSP conference series, 1998-2010.

[WPP+05] M. Woodside, D.C. Petriu, D.B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). In *Fifth International Workshop on Software and Performance (WOSP'05)*, pages 1–12, Palma, Spain, July 2005. ACM.

[WWT02] Wei Wei, Bing Wang, and Don Towsley. Continuous-time hidden Markov models for network performance evaluation. *Perform. Eval.*, 49(1-4):129–146, 2002.

[WY07] Hua Wang and Jing Ying. Toward runtime self-adaptation method in software-intensive systems based on hidden Markov model. *Computer Software and Applications Conference, Annual International*, 2:601–606, 2007.

[XML] The Extensible Markup Language (XML). `http://www.w3.org/XML`.

[XSL] The Extensible Stylesheet Language Family (XSL). `http://www.w3.org/Style/XSL`.

[XWP03] J. Xu, M. Woodside, and D. Petriu. Performance analysis of a software design using the UML Profile for Schedulability, Performance and Time. In *TOOLS*, volume 2794 of *LNCS*, pages 291–310, 2003.

[YHZ+09] Jie Yang, Gang Huang, Wenhui Zhu, Xiaofeng Cui, and Hong Mei. Quality attribute tradeoff through adaptive architectures at runtime. *Journal of Systems and Software*, 82(2):319–332, 2009.

[ZFGH00] A. Zimmermann, J. Freiheit, R. German, and G. Hommel. Petri net modelling and performability evaluation with TimeNET 3.0. In *TOOLS*, volume 1786 of *LNCS*, pages 188–202, 2000.

[ZKSZ12] Pooria Zamani, Mohammad Kayvanrad, and Hamid Soltanian-Zadeh. Using learned under-sampling pattern for increasing speed of cardiac cine mri based on compressive sensing principles. *EURASIP Journal on Advances in Signal Processing*, 2012(1):82, 2012.

Acronyms

CADA Collect, Analyze, Decide, Act loop

CSM Core Scenario Model

CT-HMM Continuous Time Hidden Markov Model

D-KLAPER Dynamic KLAPER

DARPA Defense Advanced Research Projects Agency

DOM Document Object Model

GSPN Generalized SPN

HMM Hidden Markov Model

IDC Index of Dispersion for Counts

KLAPER Kernel LAnguage for PErformance and Reliability analysis

LQN Layered Queueing Network

MAP Markov Arrival Process

MAPE-K Monitor, Analyze, Plan, Execute, Knowledge structure

MARTE The UML profile for: Modeling and Analysis of Real-Time and Embedded systems

MDE Model Driven Engineering

MMPP Markov Modulated Poisson Process

MOF MetaObject Facility

OMG Object Management Group

PUMA Performance by Unified Model Analysis

QN Queueing Network

QoS Quality of Service

SOA Service Oriented Architecture

SPN Stochastic Petri Net

UML Unified Modeling Language

XML eXtensible Markup Language