UNIVERSITÀ DEGLI STUDI DI ROMA
"TOR VERGATA"

SCUOLA DI DOTTORATO
Dottorato in Informatica e Ingegneria dell'automazione – XXII ciclo

Tesi di Dottorato

A Model-Based Approach to Performance and Reliability Prediction

Enrico Randazzo

Tutore
Prof. Vincenzo Grassi

Coordinatore del corso di dottorato
Prof. Daniel Pierre Bovet

ANNO ACCADEMICO 2009-2010

Contents

Acknowledgements **x**

1 Introduction **1**
- 1.1 Software systems . . . 1
- 1.2 Quality analysis . . . 2
- 1.3 Book structure . . . 6

2 Software engineering processes and Quality prediction **7**
- 2.1 Modern software development methodologies . . . 7
- 2.2 A very early quality prediction . . . 14
- 2.3 Prediction at design time . . . 15
- 2.4 Prediction at runtime . . . 18
- 2.5 Prediction and feedback . . . 18

3 Performance and reliability prediction models **21**
- 3.1 What kind of quality prediction? . . . 21
  - 3.1.1 Time performances . . . 21
  - 3.1.2 Reliability . . . 22
- 3.2 DTMC . . . 23
- 3.3 LQN . . . 27
- 3.4 Simulation . . . 30

4 Model Driven Approaches **35**
- 4.1 What is MDA . . . 35
- 4.2 Many ways to use MDA . . . 40
- 4.3 MDA today . . . 42
  - 4.3.1 Applications . . . 42
  - 4.3.2 Tools . . . 43
  - 4.3.3 The Eclipse world . . . 45

5 The KLAPER approach: basic concepts **47**
5.1 The intermediate meta model concept applied to performance and reliability analysis . . . 47
5.2 The meta model . . . 49
5.2.1 An overview . . . 49

6 KLAPER: syntax and semantics **55**
6.1 Meta model packages . . . 55
6.1.1 The probability package . . . 55
6.1.2 The expr package . . . 57
6.1.3 The core package . . . 59
6.2 Meta model semantics . . . 73
6.2.1 A short ASM introduction . . . 73
6.2.2 Functions for the KLAPER ASM . . . 75
6.2.3 Auxiliary functions . . . 78
6.2.4 Rules for KLAPER ASM . . . 82

7 Transforming to DTMC **95**
7.1 The DTMC meta model . . . 95
7.2 KLAPER to DTMC transformation, concepts . . . 97
7.3 KLAPER to DTMC transformation, implementation . . . 99
7.4 Using the DTMC meta model . . . 111

8 Transforming to SimJava **113**
8.1 The SimJava meta model . . . 113
8.2 KLAPER to SimJava transformation, concepts . . . 118
8.3 KLAPER to SimJava transformation, implementation . . . 120
8.4 Using the SimJava meta model . . . 125
8.4.1 Tool limitations . . . 126

9 Transforming to LQN **129**
9.1 The LQN meta model . . . 129
9.2 KLAPER to LQN transformation, concepts . . . 134
9.3 KLAPER to LQN transformation, implementation . . . 136
9.4 Using the LQN meta model . . . 158

10 A case study **159**
10.1 Tool integration . . . 159
10.2 Description . . . 161
10.2.1 The real input: UML . . . 163
10.3 The KLAPER representation . . . 165

10.4 The first run . . . 169
10.5 Analyzing Reliability . . . 172
10.6 Analyzing Performances . . . 174

11 Conclusions **179**
11.1 What we have done . . . 179
11.2 What we will do . . . 180
11.3 Related works . . . 181

A The Klaper dsl grammar **183**

B Use case example **189**

C Transformation Rules from KLAPER to DTMC **197**

D Transformation Rules from KLAPER to SimJava **207**

E Transformation Rules from KLAPER to LQN **211**

Bibliography **223**

List of Figures

2.1 UML and SysML intersection 14

3.1 DTMC state transition diagram for matrix P 25
3.2 Absorbing DTMC example 26
3.3 Simple LQN example 28
3.4 Multiple copies of servers. 30

4.1 MOF, Meta model and model hierarchy 36
4.2 Typical MOF and QVT interaction 38
4.3 PIM to PSM transformation 39
4.4 One of the simplest transformations: PIM to PSM. 40
4.5 (a) Transformation chain. (b) Models merging transformation. 42

5.1 Many to many MDA transformations 48
5.2 Use of an intermediate meta model to reduce the number of transformations 49

6.1 Package probability of KLAPER meta model 57
6.2 Package expr for the KLAPER meta model 58
6.3 Package core of KLAPER meta model 60
6.4 (a) Loop unfolding for a simple KLAPER Activity, (b) Loop unfolding for a KLAPER Activity with a nested behavior. 69
6.5 Example of correspondence between KLAPER ActualParam and FormalParam. 70
6.6 Example of “time consumption” for a KLAPER ServiceControl (that is a special case of an Activity) 90

7.1 DTMC hybrid meta model 96
7.2 Transformation of a KLAPER Join step into the DTMC meta model. DTMC doesn’t have a specific entity to represent join condition. . . . 102
7.3 Transformation of a KLAPER Fork step into the DTMC meta model. Each path of the fork is mapped into an external DTMC. 106

7.4 Transformation of a KLAPER Activity step into the DTMC meta model. (a) simple Activity (b) Activity with repetitions (c) Activity with a nested behavior. 108

8.1 SimJava meta model 114

9.1 LQN meta model 130
9.2 LQN representation of TaskActivityGraph (a) and EntryActivityGraph (b)) 131
9.3 Transformation of a KLAPER (open or closed) Workload into an LQN Processor and a reference Task 138

10.1 Integration of KLAPER into Eclipse and the related user interface . . 160
10.2 The KLAPER dsl Editor 162
10.3 Example scenario UML deployment diagram 163
10.4 Example scenario UML sequence diagram 164
10.5 Template structure for KLAPER representation of a cpu special Resource 167
10.6 Template structure for KLAPER representation of a network special Resource 167
10.7 Template structure for a KLAPER synchronous connector 168

List of Tables

3.1 LQN precedences 29

6.1 Normal distribution function attributes 55
6.2 Poisson distribution function attributes 55
6.3 Uniform distribution function attributes 56
6.4 Exponential distribution function attributes 56
6.5 Constant 56
6.6 Geometric distribution function attributes 56
6.7 Histogram distribution function attributes 57
6.8 Integer attributes 58
6.9 Double attributes 58
6.10 Unary attributes 59
6.11 Binary attributes 59
6.12 Resource attributes 61
6.13 Service attributes 61
6.14 FormalParam attributes 62
6.15 Workload attributes 63
6.16 Behavior attributes 63
6.17 Step attributes 63
6.18 Transition attributes 64
6.19 Start attributes 64
6.20 Wait attributes 64
6.21 End attributes 64
6.22 Control attributes 65
6.23 Branch attributes 65
6.24 Fork attributes 65
6.25 Join attributes 66
6.26 Activity attributes 67
6.27 ServiceControl attributes 68
6.28 ActualParam attributes 70
6.29 Binding attributes 71
6.30 Reconfiguration attributes 71

6.31 CreateBinding attributes 71
6.32 DeleteBinding attributes 72
6.33 Acquire attributes 72
6.34 Release attributes 72

10.1 Items descriptions 169
10.2 DTMC Reliability 170
10.3 SimJava Reliability 170
10.4 LQN tasks service time and throughput 171
10.5 LQN processors utilization 171
10.6 DTMC Reliability with the new network switch 173
10.7 SimJava Reliability with the new network switch 173
10.8 LQN tasks service time and throughput with the new network switch 173
10.9 LQN processors utilization with the new network switch 174
10.10 DTMC Reliability with the new network switch and disk caching . . 175
10.11 SimJava Reliability with the new network switch and disk caching . . 175
10.12 LQN tasks service time and throughput with the new network switch and disk caching 176
10.13 LQN processors utilization with the new network switch and disk caching 176

Acknowledgements

This work is the result of three long years spent at the university of Rome "Tor Vergata" and it is not only the result of my own effort, but its realization has only been possible thanks to some people that have been close to me during these years and now I want to say them: Thank you!

First of all many thanks to Prof. Vincenzo Grassi who has been my mentor in the fascinating world of software analysis and meta models. A big thanks also to the other people involved into the KLAPER project: Prof. Raffaela Mirandola, Federico Carbonetti and Fabrizio D'Ammassa.

Many many thanks to my mother and my sister for the patience and the support they gave me during all these three years.

And a big thanks also to all my friends that continued to be close to me even if a lot of time I was at home studying. In particular I'd like to thank Giordano, Mendy, Andrea, Paola, Daniele and Sara.

Chapter 1

Introduction

1.1 Software systems

Electronic and information systems are gaining every day more and more relevance in our world. We deal with the World Wide Web every day; we use our gps system each time we don't know the way to go to a destination we have never seen before; our cars use more electronic systems than what we suppose; our home banking system relies on a complex network of databases and application servers that are intended to protect our privacy and our money of course. From the big Boeing 787 to the smallest cellular phone there is a great number of electronic devices around us. That is, every day we base our lives on a rising number of complex systems.

The principal aim of today's systems is to simplify our life, but doing so a lot of things are demanded to those systems that inevitably become each day more complex and coupled between them. But the problem is that more complex systems are harder to design and requires more complex engineering approaches to be built. This is true in electronics, but this is particularly true for software systems on which today rely a lot of the features of systems.

If we think about how complex systems have evolved over the time we can see an evolution from simple standalone systems of twenty years ago (where procedural programming was an acceptable choice), to monolithic systems of ten years ago (where a single application was rich of features but sometimes with some problems of scalability and flexibility), to modern systems based on concepts like components composition, cloud computing and so on. We are transitioning from standalone

monolithic systems to a world of elements capable of interactions between them and where each component can be allocated on a different machine or can be moved from one environment to another one without loss of performances of the entire system.

So we have to deal with a lot of electronic devices that interact between them; their functionalities are every day more complex; inevitably the software required by these devices has a complexity that grows during time. But not only electronic devices are more complex, also pure software systems grows in complexity (think about the infrastructure needed by eBay for example to manage millions of transactions and payments every day), sometimes at an higher speed than electronics.

The need of more complex and efficient systems causes the evolution of engineering processes, but at the same time more efficient engineering processes lead to even more complex systems than before.

From a software point of view to solve these problems of design complexity a lot of engineering methodologies have been developed; we can mention the simple waterfall methodology, the more complex iterative methodology and the innovative agile methodologies. Some of them are completely obsolete (for example the waterfall methodology had already showed all its limits in the past), some other are continuously evolving (we can consider the variations of the iterative methodologies based on Object Oriented Programming or Aspect Oriented Programming for example), others are gaining the favour of engineers in some particular domains (for example think about the Agile methodologies particularly strong into the enterprise web environment).

But none of them is perfect and in particular each of them has a weakness in the field of software quality prediction, a field whose importance grows with the complexity of the system; that means that more complex is a software system, more important is to evaluate its quality to be sure to achieve the prefixed goals.

1.2 Quality analysis

So we just said that software engineers can use a multitude of different methodologies to build software systems. But if we analyze all these methodologies we can see that are all very focused on the implementation of the functionalities of the system but are weak from the point of view of the quality of the system. But what is "quality" in software engineering?

In [63] we can find different definitions of the concept of software quality, but according to our opinion the best one is that from Gerard Weinberg (see [55]) who states that "Quality is value to some person". So to see the quality of something we have to find the value that this thing has for someone; this is true for everything and thus this is true for software systems. But speaking as engineers do, to see the value that defines the quality of software we have to measure it.

Although there are different definitions of software quality everyone agrees that there are some factors that measure it. Again according to [63] a software quality factor is a non-functional requirement for a software program which sometimes is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality (read the value) of the software program. These factors are characteristics that one seeks to maximize in order to optimize software quality, that means that they are not a binary attribute. So rather then asking whether a software product "has" a factor, ask instead the degree to which it has or has not.

Software quality factors are:

- Understandability: clarity of purpose; all the stuffs related to the software system (code, documentation, etc...) must be clearly written so that they are easily understandable.
- Completeness: presence of all constituent parts, with each part (included input data and external libraries) fully developed.
- Conciseness: minimization of excessive or redundant information or processing.
- Portability: ability to be run well and easily on multiple computer configurations (different hardware but also different operating systems).
- Consistency: uniformity in notation, symbology, appearance and terminology within itself.
- Maintainability: propensity to facilitate updates to satisfy new requirements.
- Testability: disposition to support acceptance criteria and evaluation of performances.

- Usability: convenience and practicality of use (usually referred to Man-Machine Interface).
- Reliability: ability to be expected to perform its intended functions satisfactorily; this implies a time factor in that a reliable product is expected to perform correctly over a period of time.
- Structuredness: organization of constituent parts in a definite pattern.
- Efficiency: fulfillment of purpose without waste of resources such as memory, processor utilization, network bandwidth, time, etc...
- Security: ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations.

In spite of the number of factors we just saw, we will focus only on two of them: reliability and efficiency.

Reliability represents the amount of time the system is up and running without any failure. This is usually expressed in terms of a percentage computed by the ratio between the time the system runs properly and the total amount of time the system is in execution (including failures and crash periods).

On the other hand efficiency gives us a measure of what are the performances of the system. This kind of information can be used to solve problems like bottlenecks or to suggest to engineers a way to better plan the overall system architecture. According to [58] performances analysis (also called "performance engineering") aims to:

- Increase business revenue by ensuring the system can process transactions within the requisite timeframe.
- Eliminate system failures requiring scrapping and writing off the system development effort due to performance objective failures.
- Eliminate late system deployment due to performance issues.
- Eliminate avoidable system rework due to performance issues.
- Eliminate avoidable system tuning efforts.

- Avoid additional and unnecessary hardware acquisition costs.
- Reduce increased software maintenance costs due to performance problems in production.
- Reduce increased software maintenance costs due to software impacted by ad hoc performance fixes.
- Reduce additional operational overhead for handling system issues due to performance problems.

Reliability and performances are very important and even considering only these two factors of software quality we can obtain a lot of benefits from the point of view of the efficiency of the system.

The biggest problem with performances and reliability is that you have to implement your system and run it to see if it meets its quality requirements; that means that during the analysis and design phase of your project you can't be able to understand if it is good enough. There are a lot of best practices to help you maintaining a sufficient level of software quality (for example consider the iterative approach where each iteration has its own test phase to check before proceeding), but none of them is able to give you a measure of the goodness of your work before writing a single line of code.

To address the problem of software quality prediction a wide number of methodologies has been developed over the years whose aim is to give to engineers a measure of how their work is well done. Typically these methodologies use simulators or analytical solvers and range from software performances computation to reliability evaluation and so on. The matter now is that these methodologies are usually very far from typical analysis and design methodologies and tools and frequently software engineers don't know them or how to apply them because they don't have the specific knowledge for this kind of instruments.

As we will see later to fill the gap between software analysis and design methodologies and quality prediction methodologies we can use a particular approach called *model driven development*.

1.3 Book structure

This work is structured in a way that can give a strong theoretical base to the reader before proceeding to the inner core of the topic. In chapter 1 we gave a presentation of the environment where we move; in chapter 2 we will present the motivation of our work; in chapter 3 we will discuss some performance and reliability resolution models presenting their core structure and some related tools. In chapter 4 we will describe MDA (Model Driven Architecture), a software development approach promoted by OMG (Object Management Group) that implements the philosophy of model driven development and that is specifically designed to perform models transformations (we will see later what that means). In chapter 5 we will present a way to fill the gap between the design world and the quality evaluation world using the MDA approach. In chapter 7 we will present an implementation of a model transformation to the DTMC (Discrete-Time Markov Chains) reliability evaluation model. In chapter 8 we will present an implementation of a model transformation to another reliability evaluation model based on simulation, this model is then expressed using the SimJava library (with the SimJava based model we can also do performance evaluation but this feature is still under development). In chapter 9 we will see an implementation of a model transformation to the LQN (Layered Queueing Networks) timing performance evaluation model. Then in chapter 10 we will see how all the previous developed transformations are integrated into the Eclipse environment and we will use such a tool in a typical application scenario. To end in chapter 11 we will discuss our conclusions.

Chapter 2

Software engineering processes and Quality prediction

2.1 Modern software development methodologies

As stated in [62] a software development methodology refers to the framework that is used to structure, plan and control the process of developing a software system. A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. One system development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kind of projects, based on various technical, organizational, project and team considerations.

However in chapter 1 we said that all of these methodologies have some weaknesses into the field of software quality. Before seeing how those weaknesses can be avoided we will review some of these methodologies to understand how they work and the limits they have about software quality evaluation.

Every software development methodology (see again [62]) is characterized by its own approach to software development. But however there is a set of more general approaches, which are developed into several specific methodologies. These approaches are:

- Waterfall: linear framework type.
- Prototyping: iterative framework type.

- Incremental: combination of linear and iterative framework type.
- Spiral: combination of linear and iterative framework type.
- Rapid Application Development (RAD): iterative framework type.
- Rational Unified Process: iterative and very versatile framework type.
- Agile software development: very different from the others because focused on code and developers.

Waterfall Model

The Waterfall model is a sequential development process, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing, integration and maintenance. Basic principles of the Waterfall model are:

- Project is divided into sequential phases, with some overlap and splashback acceptable between phases.
- Emphasis is on planning, time schedules, target dates, budget and implementation of an entire system at one time.
- Tight control is maintained over the life of the project through the use of extensive written documentation as well as through formal reviews and approval by the customer occurring at the end of most phases before beginning the next phase.

It is clear that the Waterfall model has a big problem with requirements change management. Because all is produced with one single flow, no one can change a single requirement because the requirement analysis phase occurs only once into the entire software development life cycle.

By the way, from a software quality point of view you can't be able to evaluate performances or the reliability of the system until you have completed the code phase and when you could have the needed information they are completely useless because you can't modify your requirements according to the obtained results (because analysis and design phases are already completed).

Prototyping

Software prototyping is the set of activities executed during software development of "prototypes", incomplete versions of the software program being developed. Basic principles of prototyping are:

- It is not a standalone and complete methodology, but rather an approach to handling selected portions of a larger and more traditional development methodology.
- It attempts to reduce project risks by breaking a project into smaller parts simplifying the development process.
- Users are involved throughout the process; this increases the user acceptance of the final product.
- Small mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the user requirements.
- It is possible in some cases to evolve from a prototype to the working system rather then discarding the prototype.
- A basic understanding of the fundamental business domain is necessary to avoid solving the wrong problem.

The Prototyping approach is better than the Waterfall approach because it uses iterations to refine the prototype until it meets the specified requirements, but for software quality it suffers the same problem of the previous methodology, you have to realize (that means you have to write code) your application and run it to know if performances and reliability goals are achieved.

Incremental

Incremental methodology is a combination of linear and iterative development methodologies with the primary objective of reducing inherent risks by breaking the project into smaller segments and providing more ease-of-change during the development process. Basic principles of incremental development are:

- A series of mini-Waterfalls are performed; all phases of the Waterfall development model are completed for a small part of the system before proceeding to the next increment.
- All the requirements are defined before proceeding to the mini-Waterfall increment.
- Sometimes a Waterfall approach is followed until the design phase, than from the coding phase a prototyping approach is followed.

Again with the Incremental approach we have a combination of the problems already seen with previous methodologies: there are some problems in the requirements change management and the system have to be implemented and run to evaluate software quality.

Spiral

The Spiral methodology is an iterative approach that cycles over four main phases (Analysis, Evaluation, Development, Planning) to build the software system in an incremental way. Basic principles are:

- Focus is on risk assessment and on minimizing project risks by breaking it into smaller segments and facilitating changes during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
- Each cycle involves a progression through the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from an overall concept of operation document down to the coding of each individual program.
- Each trip around the spiral traverses four basic quadrants: determine objectives, alternatives and constraints of the iteration; evaluate alternatives and identify and resolve risks; develop and verify deliveries from the iteration; plan the next iteration.
- Begin each cycle with an identification of stakeholders and their needs; end each cycle with review and commitment.

Compared to previous methodologies we have a clear improvement concerning the change management process but as usual the system has to be completely built to see its quality performances.

Rapid Application Development

Rapid Application Development is a software development methodology which involves iterative development and prototyping. Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991. Basic principles are:

- The key concept is a fast development and delivery of a high quality system at a relatively low cost.
- The project is split into smaller segments to reduce project risks and to provide more ease-of-change during the development process.
- It aims to produce high quality systems using iterative prototyping, active users involvement and advanced tools (like CASE tools, graphical user interfaces and so on).
- If the project starts to slip, emphasis is on reducing requirements to fit the time frame, not in increasing the deadline.
- Active users involvement into the development process is imperative.
- Iteratively produces production software and not only a throwaway prototype.
- It produces documentation necessary to facilitate future development and maintenance.
- It can be used standard system analysis and design techniques.

Once again we have a good management of the system development from a functional requirements point of view but we lack support for software quality evaluation (as usual we need to write and build code to see whether software quality requirements are met).

Rational Unified Process

Rational Unified Process is an iterative methodology developed by Rational Software and it not defined a single process, rather it defines an adaptable framework that leads to different processes depending on the project and on the organization that applies it.

A project life cycle consists of four phases:

- Inception: can be considered a feasibility study.
- Elaboration: in this phase software engineers do a domain analysis and project the overall software architecture.
- Construction: this is the phase where true developments are made. At the end of this phase we have the first release of the system.
- Transition: the system is moved from the development environment to the customer for final refinements.

Each phase consists in some “discipline”:

- Business modeling.
- Requirements.
- Analysis and design.
- Implementation.
- Test.
- Deployment.

where the relevance given to each discipline varies according to the phase where it is executed (but all disciplines are present in each phase).

Due to its versatility Rational Unified Process today is used for a wide range of projects by different companies. Thanks to the high level of iteration change management is very well supported but for software quality we meet again the same problems found in previous methodologies.

Agile software development

Agile software development refers to a group of software development methodologies based on the iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto (see [20]) was formulated. Basic principles are:

- people and interactions are more important than processes and instruments.
- It is more important to have runnable code than documentation.
- Direct collaboration with the customer is more important than the business contract.
- The development team should be authorized to suggest modifications to the project to actively react to changes.

This methodology is very practical development oriented and focuses its effort to changes management, customer satisfaction and an effective code development. But again, there is no way to evaluate software quality before obtaining the code, even if in this kind of approach that is very "code centric" this is not a very big problem because code is one of the first results of the methodology.

As we can see a lot of software development methodologies have been developed during the years, each with its specific features, more or less generic, but all have one limitation regarding software quality evaluation: the system, or one of its iterations, has to be built (that means that we need some code that compiles and runs) in order to collect data to use for the overall system performance and reliability evaluation.

Our aim is precisely to show how software engineers can consider quality factors like performance and reliability at any stage of the software development process (whatever it is) without the final system and without writing a single line of code. More precisely our focus will be about software performance and reliability evaluation at design time.

2.2 A very early quality prediction

As we just said our aim is to evaluate software performances and reliability even before developing the code, but when do we start to need such an evaluation? The answer is quite simple: as soon as possible. That means that even during requirements definition someone could be interested in software quality.

Usually during the requirements analysis phase software quality evaluation is not a real priority because in this phase we are more interested in understanding which are the performances and reliability requirements of our system rather than measuring them, but someone could have such a need.

It can seem very difficult to evaluate software quality at requirements analysis time but if we use structured languages for the definition of requirements like SysML we can see that it is really close to what we will see in the next section about software quality evaluation at design time.

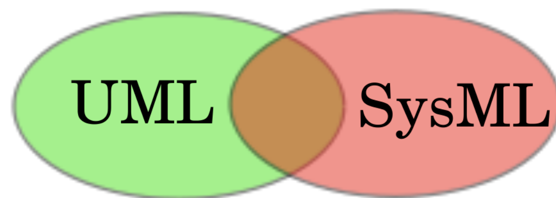

Figure 2.1. UML and SysML intersection

SysML (see [40]) is a language built as an evolution of UML but for system requirements definition. It can use almost all UML diagrams and in addition it introduces some new ones (for example the Blocks diagram). Deriving from UML also SysML can define its own set of stereotypes and as we will see later these stereotypes can be used to enrich diagrams with performance and reliability input annotation. Once we have an annotated input SysML model (it can be a block diagram, a class diagram and so on) we can apply all the consideration that we will see in the next section.

Obviously we are at very early stage into the development process, therefore we can't use precise measurements and the obtained results must be considered at a very high level and with a high error margin; however results can be used to give

to engineers a general idea of what would be the behavior of the system and what could be the failure points of the system architecture.

2.3 Prediction at design time

Whatever software development methodology we use (see chapter 1 for a brief introduction to this topic) the design phase (here with the term "design" we refer to analysis and design activities) is very important because this is the phase were all the system is really built; the coding phase is only a conversion, preferably done automatically, from project to code and should not have a big relevance.

Starting from the requirements analysis and during all the design phase software engineers (implied roles are software architects and software designers) analyze requirements, study the application domain, make a software architecture that will be the infrastructure of all the software system and then they design each particular of the system, from external communication interfaces to internal communications between software components, from the man-machine interface to the interaction with the chosen operating system, from the layers that compose the system to the way the log has to work and so on; in short every particular of the system should by decided during the design phase.

The problem is that if we look at actual software methodologies as they are applied in the industry real world we can see that usually the design phase is underestimated running directly to the code phase with obvious catastrophic consequences in case of problems (due to requirements changes or to design errors). The fact is that actual design approaches are very functional requirements oriented; that means that the main concern for engineers is to satisfy functional requirements. But the more complex systems become the more functional requirements are not the only objective to achieve, on the contrary when a system is very complex non functional requirements such as performances and reliability play a crucial role because they can deeply influence changes into the design and even in software requirements.

So to evaluate quality indexes during the design phase is a crucial activity before turning the project into code because this last activity could be completely useless if some major changes of the system are needed to achieve non functional requirements.

To solve this problem in the last years a number of methodologies have been developed to help people to evaluate performance and reliability values during the

design phase. In particular as reported in [50] we can categorize the different approaches as follows:

- Queueing network based methodologies.
  - Methodologies based on the SPE[1] approach.
  - Architectural pattern-based methodologies.
  - Methodologies based on trace analysis.
  - Uml for performance[2].
- Process-Algebra based approaches.
- Petri Net based approaches.
- Methodologies based on simulation methods.
- Methodologies based on stochastic processes.

Even if [50] is quite outdated the considerations done about methodologies categories are still valid.

- The preferred input method is usually UML with one of its profiles[3] as extension used to annotate performance and reliability values.
- Queueing networks are the preferred performance model.
- Additional information is needed to augment the base system model representation with performance and reliability information.
- Feedback management is not very well managed (see section 2.5).
- There is much need of automated tools that support engineers in performance and reliability evaluation.

[1] SPE stands for Software Performance Engineering and was a method introduced by Smith in [51].

[2] methodologies based on uml notation as input to add relevant information to the input model

[3] Usually UML-SPT (see [43]) or the MARTE profile (see [42].

From the points just seen we can find a great amount of information to think about, but there are three main issues: need of software quality evaluation models, additional information needed by quality models, need for tools.

To evaluate software quality a certain number of quality evaluation models have been developed until now. Such models range from queueing networks (with all their variants), to Petri nets, to stochastic models, to process algebras, to simulation models. Typically each model is well suited for a particular type of analysis; there are models specific for performance evaluation (for example queueing networks) and models specific for reliability evaluation (for example simulation, etc. . .), but none of them is useful for evaluation of all aspects of software quality. So engineers have to choose the right model for the quality index they want to evaluate and usually they have to use more than one single model on the same design project to address all needed quality aspects. Moreover not all models are suitable all kind of systems; some are good for web based systems, some are good for embedded systems, some other are more general and can be adapted to a more variety of systems. However the choice of the right models has to take into account all these factors.

But choosing the right quality evaluation model is not the only problem. To compute software quality we need to provide additional information, to someone or something, that will be used in addition to the design of the system; in other words the design of the system tells what is the behavior of the system but don't tell for example what are the execution times of some particular actions or the percentage of active time before a failure of a particular component; such measures have to be provided to the quality evaluation model based on measurements from similar system already completed, estimates, black box elements compositions and so on, but have to be provided in addition to the system design (and here then raises the problem about how this information must be provided to the design model, for a deeper understanding of this topic see appendix **??**).

The last issue is about the need of tools. Usually performance and reliability are not taken into account by engineers simply because they don't know them or, if they know the topic, they don't know how to use such models in a productive way considering the time they have to design the system. To fill this gap usually quality evaluation models are provided with a variety of tools that are in charge of simplifying designers life.

As already said quality evaluation at design time is the main topic of this work, therefore in the following chapters we will see how the problems discussed until now can be solved (or better we will see one of the possible solutions).

2.4 Prediction at runtime

Even if it is not within the scope of this work, it is useful to know that the same techniques used at design time by software engineers, can be used by the system itself.

The system can be built with a knowledge of itself. This knowledge can be used at runtime to evaluate at precise instants the performances and the reliability of the system; such measures can be used to modify the behavior of the system itself to better suit its goals (for example think about a load balancing accomplished by a web proxy according to some QoS rules or to a distributed system that changes the deployment according to the availability of nodes).

Giving to the system the capacity to evaluate its software quality and reacting in the best way adds to the system the capacity of self adaptation to situations. Self adaptive systems is a very interesting research field (see [19]) and that is giving a great number of useful results to the engineering community.

2.5 Prediction and feedback

Software quality evaluation (or prediction) can be done either at design time than at runtime and it is interesting to note that there are some research areas that deal with the way the results of these evaluations are used; in other words there are some researches about how to use the feedback from quality evaluation.

In the rest of this work we will focus on performances and reliability prediction at design time and we will see how engineers can use results obtained from evaluations to take decisions over the overall system structure. We will concentrate on methods to obtain measures and we will touch only marginally the way these measures are used. Actually there are ongoing researches that are studying how software quality

measures can be used in a systematic (read automatic) way to reflect appropriate changes back to the system (either at design time than at runtime).

Even if feedback implementation at design time and at runtime has different problems (basically completeness of the information for the former and stability and overhead for the latter) some methodologies have been proposed like those in [64] and [46])

However this is only for informational purposes since the main topic of this work is about software quality prediction at design time and we don't wont to go deeper into this argument (even if it is really interesting).

Chapter 3

Performance and reliability prediction models

In chapter 2 we saw that engineers need some methods to evaluate software quality at design time, but at this point we have some open questions: which indexes can we use to evaluate software quality? And which quality models can we use to evaluate these indexes? In this and the following sections we will try to give some answers to these questions.

3.1 What kind of quality prediction?

In section 1.2 we told that software quality is the value that the software can have for someone and we enumerated a number of factors that can be evaluated to express the degree of this value. However from a practical point of view some of the factors presented are not very significant to engineers during the design phase, so we will concentrate, as already anticipated, only on two of them: performances and reliability.

3.1.1 Time performances

Performance indexes tell us how fast is the system, how it performs; in other words when we talk about performance indexes we are talking of activities execution time. Evaluating the execution time of the components of a system is very useful because from this metric we can derive very important indexes like throughput and

utilization.

Throughput shows how many requests per second a component can satisfy; on the other hand utilization shows the percentage of time a single component spent serving requests from users or from other components. Combining these two indexes we can know a lot of things about the system, but the most important thing is that analyzing throughput and utilization we can decide if non functional requirements (in this case non functional timing requirements) have been satisfied and if not we can understand where to apply changes to the system to meet the required goals (for example utilization and throughput are very useful to identify very annoying problems like bottlenecks). Just to give some examples if we have a low throughput with a low utilization probably we oversized some components of our system; instead if we have an high throughput with an high utilization (near 99% for example), that means that probably we have found a bottleneck with a possible loss of requests.

3.1.2 Reliability

In [29] reliability is defined as "the ability of a system or a component to perform its required functions under stated conditions for a specific period of time"; so reliability defines how long the system runs properly into its environment before a failure. This is a very important information to take into account because gives us the capacity to evaluate the robustness of our system. For example for a home banking web systems we could have some requirements stating that the system have to be up and running seven days a week (that means that when in power-on state the system can never be down, therefore the required reliability must be equal to 100% in the related time interval) or for example in an avionic system we could have some requirements (typically derived from an avionic certification) that the system can't have a failure probability greater than 0.01 per hour because otherwise we could loose lives and fail the mission.

In [2] it is introduced the concept of dependability of a system as the ability to avoid service failures that are more frequent and more severe than is acceptable. Always in [2] dependability is defined as an integrating concept that encompasses the attributes:

- *availability*: readiness for correct service.
- *reliability*: continuity of correct service.

- *safety*: absence of catastrophic consequences on the user(s) and the environment.
- *integrity*: absence of improper system alterations.
- *maintainability*: ability to undergo modifications and repairs.

So reliability is an attribute of dependability that expresses the continuity of correct service; usually it is computed starting from the failure rate of system components and can be evaluated as the probability that the system correctly performs its required functions under stated conditions for a specified period of time (see [25]) or as the probability that the system successfully completes its task when it is invoked (also known as “reliability on demand”, see [30]).

Now that we know what we want we will see some different models to use to evaluate performance ad reliability. In this chapter we will give a theoretical presentation of these models while in the following chapters we will show an effective way for using them in a model driven environment.

3.2 DTMC

The acronym DTMC stands for Discrete Time Markov Chains; in this section we will give an overview of DTMC; for a more detailed description see [21], [27] and [5].

A *Markov chain* (see [56]) is defined as a Markovian process[1] with a finite states space S, where states transitions occur randomly in discrete steps. One important rule in Markovian processes is that the probability distribution for the system state at the next step (and at all future steps) only depends on the current state of the system and not on its state at previous steps. In a more formal way a stochastic process $\{X_0, X_1, \ldots, X_{n+1}\}$ with consecutive observation points $0, 1, \ldots, n+1$ is a DTMC if its conditional *pmf*[2] satisfy the following property for each $n \in \mathbb{N}_0$ and for

[1] More details on morkovian processes can be found in [57]

[2] Probability Mass Function, see [59]

each $s_i \epsilon S$:

$$P(X_{n+1} = s_{n+1} \mid X_n = s_n, X_{n-1} = s_{n-1}, \ldots, X_0 = s_0) = P(X_{N+1} = s_{n+1} \mid X_n = s_n) \tag{3.1}$$

Since the system state changes randomly it is not possible to predict its exact state in the future; however we can consider the statistical properties of the system and use these properties to study the behavior of the system itself. For example Markov chains can be used to represent the evolution of the states of a software system that may be or not in a failure state, giving us the possibility to have some statistical information about reliability.

From a practical point of view to study DTMC we have to know the following characteristics:

- The states space S.
- The transition probability from a state i to the next state j denoted as p_{ij}. Thanks to 3.1 this probability depends only on the states i. It must hold: $\sum_j p_{ij} = 1$.
- The n-dimensional probability vector $\mathbf{u}$ (n is the number of possible states) which represents the initial distribution (each element i of the vector represents the probability to be at state i at instant 0 and the sum of all elements of the vector must be equal to 1).

Given a states space S and the transition probabilities p_{ij}, it is possible to describe a Markov chain with a non-negative stochastic matrix $\mathbf{P}$:

$$\mathbf{P} = \begin{bmatrix} p_{00} & p_{01} & p_{02} & \cdots \\ p_{10} & p_{11} & p_{12} & \cdots \\ p_{20} & p_{21} & p_{22} & \cdots \\ \cdots & \cdots & \cdots & \ddots \end{bmatrix} \tag{3.2}$$

where the sum of the elements of each line of the matrix P have to be equal to one. From the matrix P we can obtain the *state transitions diagram*. For example if we have the matrix

$$\mathbf{P} = \begin{bmatrix} 0.75 & 0.25 \\ 0.5 & 0.5 \end{bmatrix}$$

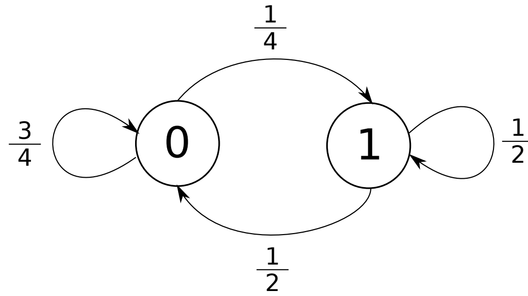

Figure 3.1. DTMC state transition diagram for matrix P

we can obtain the diagram reported in figure 3.1.

If $\mathbf{P}$ is the transitions matrix for a Markov chain, the probability to go from the state i to the state j in exactly n steps is given by the $ij-th$ entry of the matrix $\mathbf{P}^n$. So if we consider the u vector previously presented we can say that if P is the transition matrix for a Markov chain and u is the probability vector that represents its initial distribution then the probability that the chain will be at state i after exactly n steps is given from the $i-th$ entry of the vector

$$u^{(n)} = u\mathbf{P}^n \tag{3.3}$$

From the reliability point of view there is a very interesting Markov chains type called *Absorbing Markov chains*. An absorbing Markov chain has at least one state s_i (it is called *absorbing state*) that can't be left (that means that $p_{ii} = 1$) and that can be reached from any other state in a finite number of steps. Any state that is not absorbing is called *transient state*. Absorbing Markov chains are useful for evaluating reliability because fault conditions, that are conditions from which the system can't go out, can be represented with absorbing states. But before we can use such particular Markov chains we have to introduce some other concepts.

Given any Markov chain, the canonical form of its transition matrix P can be obtained renumbering the states so that transient states are placed before absorbing states; supposing to have r absorbing states and t transient states the canonical matrix is:

$$\mathbf{P} = \begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \tag{3.4}$$

where:

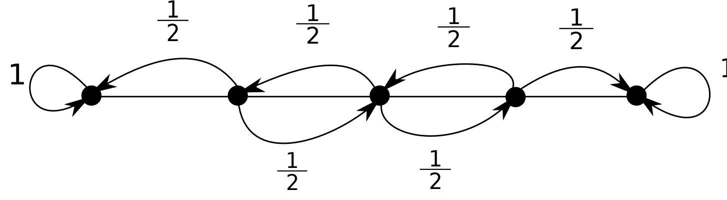

Figure 3.2. Absorbing DTMC example

- $\mathbf{I}$ is an identity matrix $r \times r$.
- $\mathbf{O}$ is a zero matrix $r \times t$.
- $\mathbf{R}$ is a non-zero matrix $t \times r$.
- $\mathbf{Q}$ is a $t \times t$ matrix.

and the first t states of the matrix are transient states while the last r are absorbing.

If $\mathbf{P}$ is in canonical form then $\mathbf{P}^n$ has the form:

$$\mathbf{P}^n = \begin{bmatrix} \mathbf{Q}^n & * \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \tag{3.5}$$

where * represents a $t \times r$ matrix and elements of $\mathbf{Q}^n$ are the probability of being at each transient state after n steps. Increasing n this matrix contains values smaller and smaller; indeed there is a theorem that states that in an absorbing Markov chain the probability that the process will be absorbed is equal to 1 (that means that $\mathbf{Q}^n \to 0$ for $n \to +\infty$).

Returning to the canonical matrix it can be shown that for an absorbing Markov chain the matrix $(\mathbf{I} - \mathbf{Q})^{-1} = \mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \ldots$ called *fundamental matrix* for the Markov chain represented by $\mathbf{P}$, plays an important role. Indeed, the $ij - th$ entry of $(\mathbf{I} - \mathbf{Q})^{-1}$ represents the expected number of visits to the state s_j assuming that s_i is the initial state. The initial state is included in the count if $i = j$.

Knowing $(\mathbf{I} - \mathbf{Q})^{-1}$ we can see why DTMC are so useful for evaluating of reliability: if b_{ij} is the probability that an absorbing Markov chain, whose initial state is s_i, will be absorbed into the state s_j than we can call $\mathbf{B}$ the matrix $t \times r$ which has as entries the various b_{ij}; $\mathbf{B}$ can be computed as in [8]:

$$\mathbf{B} = \mathbf{NR} \tag{3.6}$$

where **N** is the fundamental matrix and **R** is the same as in 3.4.

If we consider reliability as the probability that a system will accomplish its duties without any failure, than we can represent a system with an absorbing DTMC with at least two different absorbing states: one that represents the correct end of system execution without any failure and one that represents the system when some failure happens; in that case from **B** we can obtain all the information we need to evaluate the reliability of a software system.

3.3 LQN

As stated in [22] the Layered Queueing Network (LQN) model is a canonical form for extended queueing networks with a layered structure. The layered structure arises from servers at a given layer making requests to servers at lower layer as a consequence of a request from a higher layer. LQN have a more complex structure if compared with DTMC so we will discuss only the model architecture without any reference to the mathematical concepts that lie behind it.

In LQN (for an example see figure 3.3) the basic building blocks are *tasks* that run over processors (each task runs over a single processor) consuming execution time of the processor itself (processors follows specific scheduling policies to choose which of the tasks the rely in it to execute). Each task provides services through some entry points called *entry*. Each entry can provide its service in two different ways:

1. Phases: the activities of the entry are executed in different phases. Phase one is a *service* phase and is used to reply to the service request, after phase one is completed and a response is sent back to the requester, remaining phases are executed one after another. Phases consume execution time on processors and can make requests to other entries.

2. Activities graph: the requested service is provided executing the activities linked into an activities graph (activities are defined below).

In point 2 we introduced the concept of *activity*. Activities are the lowest-level of specification in the performance model; they consume time on processors, can call entries (that reside on the same task or on other tasks) and are linked together to

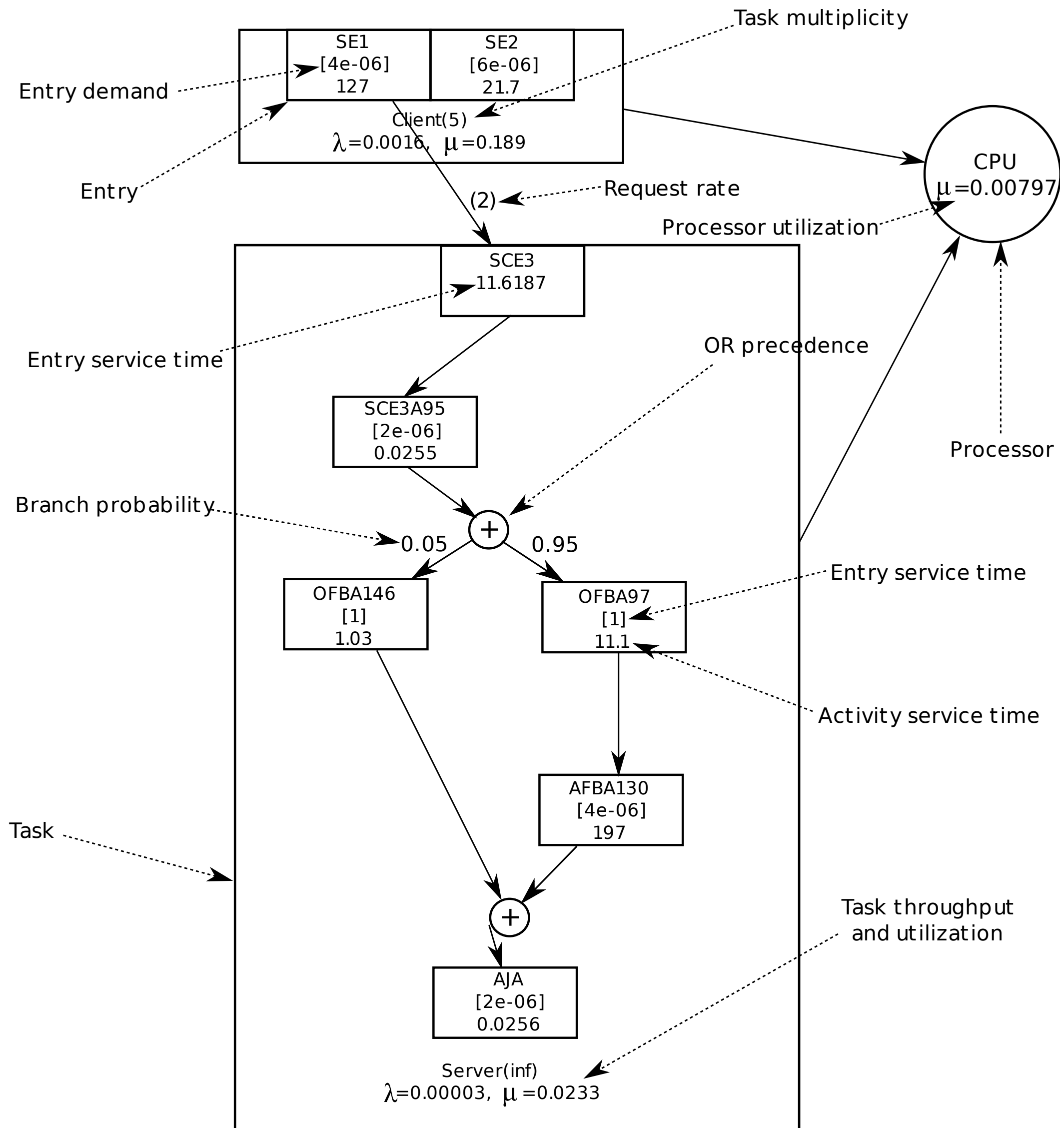

Figure 3.3. Simple LQN example

form an execution graph through some special elements called *precedence*. Precedences are special nodes into the activities graph that determine the execution flow of activities; available precedences are reported in table 3.1

We just said that activities can request services to entries; there are three different requests types:

- Rendezvous requests: these are typical synchronous calls where the requester

Name	Description
Sequence	Transfer of control from activity to join-list or from fork-list to activity.
And-Join	A synchronization point for concurrent activities.
Or-Join	The merge of two different branches.
And-Fork	Start of concurrent execution.
Or-Fork	A branching point where one of the paths is selected.
Loop	Repeat the activity an average of n times.

Table 3.1. LQN precedences

is blocked until the server has completed its work and has sent back a response.

- Forwarding requests: the request is sent to a server but the response to the requester is sent from a different server (to which the request is forwarded). The requester is blocked until the response is received.

- Send-no-reply: these are typical asynchronous requests where the requester make the call and then it goes forward without waiting for any response.

The last topics about LQN are *multiplicity* and *replication*. There is a subtle difference between the two concepts because with the word “multiplicity” LQN refers to a configuration of a single service center where there is a single queue that is served by multiple servers (see figure 3.4a). Instead with the word “replication” LQN refers to a single service center where there are multiple queue served by multiple servers (see figure 3.4b).

So, as usual for queueing networks, all the LQN world turns around the concept of entities (jobs, in the usual queueing network terminology) consuming processors’ execution time, but LQN adds the concept of a layered structure to the model of the system. Typically queueing networks are used to compute execution times and therefore timing performances (throughput and utilization) but the layers decomposition of LQN models gives the possibility to better inspect the system and to easily locate design problems like unused components, bad patterns, bottlenecks and so on.

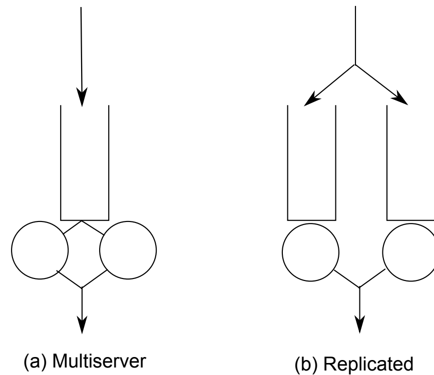

Figure 3.4. Multiple copies of servers.

3.4 Simulation

Until now we have seen only quality models that can be solved (at least in principle) analytically; that means that there is a strong mathematical foundation behind each model that can be used to compute results without running the real system. In such cases indexes (such as throughput, utilization or the overall failure rate) can be evaluated in an analytical way solving groups of equations. But analytical computation is not the only way to obtain the required indexes. When the system is not yet available (for example when you don't have the system because you are designing it) or when the system is too complex to be solved mathematically (because cpu processing power is limited) an alternative solution is that of building a *simulator*.

According to [61] simulation is the imitation of some real thing, state of affairs or process. The act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system.

For software systems simulation is the practice of building a sort of prototype of the model that doesn't implement its functional requirements but is sufficient to simulate the behavior of interest; when such a simulator is ready you can simply run it and sit down seeing how the system would act (or better should act) if it was really implemented. Something very important to understand is that the simulator is not the real system nor in its features nor in the deployment, it is simply an object that is very close to the behavior of the system compared to the aspect under evaluation (therefore a simulator for studying an aspect of the real system is completely useless

if considered for some other aspects). There are simulators for different types of system analysis. In this work we will consider a Java[3] simulation library called SimJava.

SimJava (see [54] and [33]) is a software library capable of achieving discrete events simulations (see [9]). In SimJava the system is seen as a set of entities that interact between them; entities are linked together and communicate raising events. This event flow is managed by a main class that orchestrates all the interactions.

The main concept in SimJava is the concept of *entity*. An entity is a process that acts independently and can collaborate with other entities (or processes). From an implementation point of view each entity is a Java Thread that runs autonomously and its behaviour is realized by the implementation of its body() method. Entities are linked together using *ports* and the interaction between them occurs using *events*; to manage events the library has three main features:

- Sending an event to an entity or to an entity port (this action is done using the sim_schedule() method).
- Turning an activity into a wait state (the entity is effectively descheduled) until it receives an event (the wait is done using the sim_get_next() method).
- Managing the simulation clock running or pausing it (the method sim_process() consumes time simulating some sort of activity accomplished by the entity while sim_pause() pause the entity execution).

Each entity has a queue of received events; events are selected from this queue using some *predicates*. Predicates are a kind of conditions that act like filters on events. We can subdivide predicates into three types:

- General predicates: are used to select any event (SIM_ANY) or to select no events (SIM_NONE).
- Tag predicates: the first event is selected that has the specified tag (the tag is an integer value used as an event identifier to see what kind of event is into the entity's queue).

[3] Java is only an instrument to realize the simulator and it does not means that we can only simulate Java applications

- Source entity predicates: the selection of the event to be extracted and managed from the waiting queue is done taking into account the entity that generated the event.

Two very important aspects to be considered are how the simulation starts and its termination condition. The time that elapses from the initial instant of the simulation until it reaches a steady state is called transient or warmup period; during this time results are not very useful because they represents only a situation that occurs into the simulator but has not an equivalent into the real system (because the simulation refers only to a limited period of life of the system) so typically warmup period results are discarded. In SimJava there are several ways to define a reached transient condition:

- The number of completed events from the beginning of the simulation.
- The elapsed time from the beginning of the simulation.
- The "min-max method": is a technique to automatically determine the warmup period.

On the same way the termination condition is a condition that when met stops the simulation; you can specify a termination condition in three different ways:

- All events are completed: that means that all events have been managed.
- Elapsed time: the simulation time is ended.
- Confidence interval accuracy: the simulation ends when the mean of a measure has reached the desired accuracy.

The most relevant difference between analytical resolution models ad simulation models is that using simulation the solution is not mathematically calculated but it is based on some observed measures. Typically one simulation run is not enough to have a meaningful evaluation of the system because it refers only to one specific scenario, so to avoid this problem usually the simulation is run several times changing the random number generators seeds to have different results (simulating a more realistic scenario); when the results of all the runs are available some techniques are used to extract the (statistical) results.

In this chapter we briefly analyzed three of the possible quality analysis models that software engineers can use to evaluate software quality at design time. It is clear that to use these techniques one has to have some knowledge of the topic and this can't always be assumed for members of a software development team. Furthermore it is clear that such models are something not so close to the usual tools used into a typical design phase.

In the next chapter we will present a way to fill the gap between typical methodologies used during the design phase and the software quality evaluation models presented so far (but the same approach can be extended to every quality analysis methodology).

Chapter 4

Model Driven Approaches

So far we have stated the problem: software quality analysis is a powerful instrument considering the complexity of today software systems and should be performed as soon as possible, preferably during the design phase, but engineers typically don't have the right knowledge to use some of the many methodologies developed for this purpose and usually the models they use to design the system are very different from those used to evaluate software quality.

In this chapter we will see one of the possible solutions to this problem, the *Model Driven Approach*.

4.1 What is MDA

According to [35] (but see also [36]) a model of a system is a description of that system and its environment for some certain purpose. Model Driven Approach (or Model Driven Architecture, or Model Driven Engineering, call it whatever you want but it is always the same concept) is an approach to system development, which increases the power of models; it is *model driven* because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification of a system.

The central concept of MDA is the *model*, or better the way to define the model. MDA relies on MOF (Meta Object Facility, see [38]) that was born from the core packages of UML (see [41]) using some of its simplest elements; some of the key concepts of UML, like the concept of classes, attributes, relations (only binary),

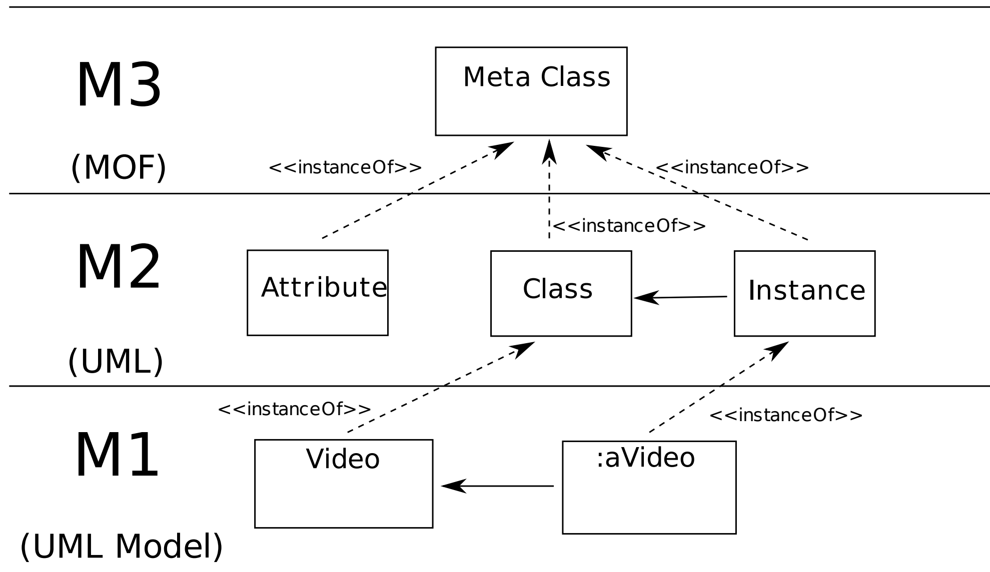

Figure 4.1. MOF, Meta model and model hierarchy

containment and so on, was taken to build something that could be used to demonstrate and define UML itself. This subset of UML was called MOF (Meta Object Facility) and its entities and "primitive" concepts allow to define not only the structure of UML, but also the structure of some other completely different models; they can even be used to define MOF itself (that is MOF is self supporting).

MOF is at the base of MDA because it allows to define meta models; but it is better if we go step by step. Referring to the figure 4.1 in MDA we have three different levels of abstraction. The first level (denoted as "M3") is MOF itself and it is also called the *meta meta model*. The entities contained in MOF (concepts called *meta meta classes*) can be used to define elements of the second level (denoted as "M2") called *meta model* that is the abstract representation of some sort of method or concept; a meta model is an instance of a meta meta model. At the third level (the lower one, denoted as "M1") called *model*, we have one of the possible instances of the entities of the second level. For example considering UML, at level M1 we have the UML model of a system; this model is an instance of the concepts contained at level M2 (where we can find entities like classes, attributes, relations, diagrams and so on) into the UML meta model; at the end of this hierarchy the UML meta

model of level M2 is only an instance of the meta meta model MOF.

So MOF can be used to define meta models and a typical meta model example is UML. But UML is not the only meta model we can define; using MOF we can define every sort of meta models from which we can instantiate every sort of model.

MOF is only one element of MDA because it allow to define meta models. The true power of MDA is given by the ability to define transformations from a meta model to another thanks to the fact that all the meta models share the same definition language: MOF. The concept of *transformation* is fundamental in MDA because once you have two meta models and you have defined some transformation rules between them you can apply those rules to every instance (that means to every model) of one of the two meta models.

The transformation process between meta models is so important that OMG has developed a dedicated language called "Query/View/Transformation", or QVT, whose specification can be found in [37]. QVT defines a language to build transformations; it has a syntax very similar to OCL (see [39]) and actually there are two main variants of it: QVT Operational and QVT Declarative. The former is an imperative language where you describe model transformations using the procedural style of an imperative programming language (for example like the C language), while the latter is based on the concept of defining relations between the source and the target meta model. At the moment QVT Operational already has some different working implementations and despite its OCL like form (not very comfortable for traditional procedural developers) it is widely adopted in the MDA environment. QVT Declarative is a completely different approach to model transformation. Its main feature is that writing the transformations in the form of relations they are automatically reversible, that means that when you write a transformation with QVT Declarative you can implement only one transformation that can be used in both ways from the meta model A to the meta model B and from the meta model B to the meta model A. On the contrary with QVT Operational you have to write one specific transformation for each way; it is clear that QVT Declarative is more powerful than QVT Operational, but this power has a cost to be paid. Indeed, these kind of transformations are harder to write if compared to QVT Operational and also the transformation engine is more difficult to build: this is evidenced by the fact that actually there isn't any mature implementation available for this transformation language.

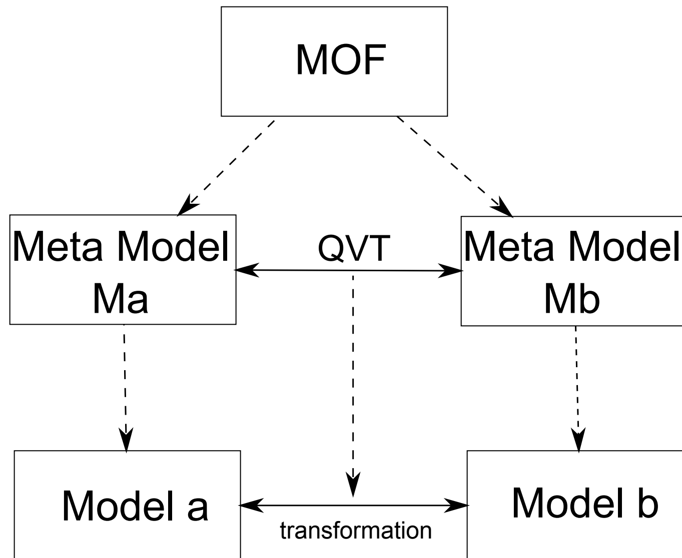

Figure 4.2. Typical MOF and QVT interaction

As we can see in figure 4.2 with MOF that can be used to define meta models and with QVT that allows transformations between meta models, now we have all what we need to transform a model into another one using the MDA way.

As already stated MDA is an approach to systems development, so OMG in [35] suggests some guidelines to be followed in building a system using MDA approaches. According to OMG, during the development life cycle the first obtained artifact is called PIM, that is Platform Independent Model; this means that the first result of developing a system using models must be a model that is completely independent of the specific platform where the final system will run; for "platform independent" usually we mean independent of the underlying hardware but it could be even independent of some software infrastructure (for example independent of the specific operating system in use). After we have obtained a PIM we should transform it into a PSM, a Platform Specific Model that can be used as a starting point for code generation (typically done automatically). See figure 4.3 for a schema of what we just said. The advantages of this kind of approach are two:

1. The PIM independence from low level details permits to architects and designers to concentrate on the structure of the system and on the domain problem to solve.

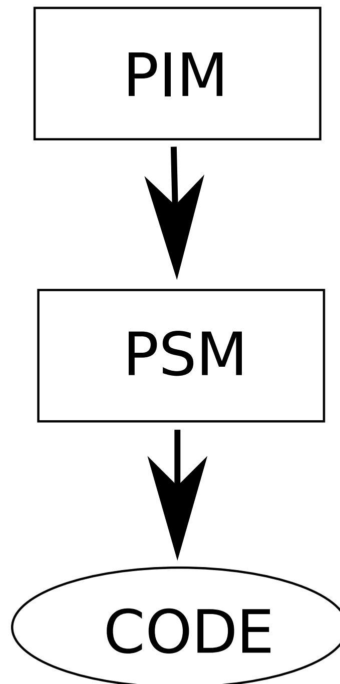

Figure 4.3. PIM to PSM transformation

2. If the PSM changes, the overall structure of the system (in that case the PIM) doesn't require changes; all you need to do is to build a new transformation from the same PIM to generate a new PSM.

Obviously this approach has a lot of advantages if compared to other methodologies that we have seen so far. A very simple and frequent example of this method is starting from an UML model (the PIM) of the system that is completely independent of the programming language to use; we can design our system without considering all the programming language constraints and then when the design is completed we can transform our UML model into a C++ model (the PSM) and then generate code, or if C++ is not the right choice and for example we need some kind of application portability we can transform to a Java model (another PSM) and then generate code without any change to our input UML model (the PIM).

The previous one was only an example of a typical MDA approach and we have to consider that it is the simplest application case; MDA can do much more powerful things. The chain $PIM \rightarrow PSM \rightarrow$ *code* of figure 4.4 demonstrates how a typical

transformation from a PIM to a PSM is realized: transformation rules from the PIM to the PSM are built starting from the analysis of the final platform where the PSM will run.

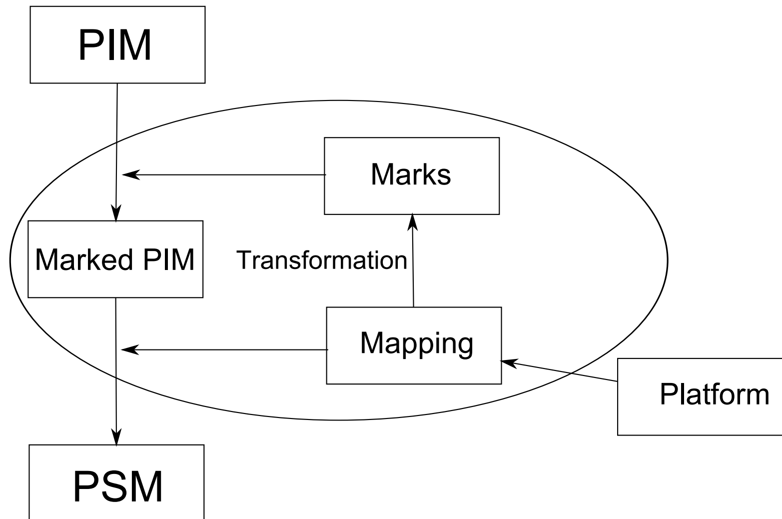

Figure 4.4. One of the simplest transformations: PIM to PSM.

4.2 Many ways to use MDA

The PIM to PSM transformation is only one of the possible applications of MDA and it is an example of a model-to-model transformation.

In MDA we can identify two different types of transformations:

- M2M: model-to-model transformations.
- M2T: model-to-text transformations.

In M2M transformations the source and the target of the transformation are models, that means that we can go from a model a conforming to a meta model Ma to a model b conforming to a meta model Mb, or we can use the same meta model as starting point and ending point of the transformation. The first kind of transformation is also called an *endogenous* transformation, while the second one is called an *exogenous* transformation (see [60]). In M2T transformations the source is a model

while the target is text; this is the typical transformation used for example when someone wants to automatically generate code from a model. Even if M2T transformations can seem out of the scope of the MDA approach that relies on models we have to understand that code, or better text, can be considered as a particular representation of a kind of model. To substantiate this thesis we have to consider that according to OMG specifications (see for example [44]) models must be stored into XMI files, where XMI is just a particular dialect of XML and therefore it is text.

Until now we have seen the $PIM \rightarrow PSM \rightarrow \mathit{code}$ paradigm where we have an input design model that is transformed to a platform specific model from which we can generate code. But as already said this is just one of the possible applications (and this is one of the simplest cases); MDA doesn't limit us in the number of transformations we can apply in sequence, nor in the kind of transformations, nor in the number of input models we can use for a single transformation. We can apply more than one single M2M transformation in sequence before applying a M2T transformation (see figure 4.5a), for example this can be done for an extension of the typical $PIM \rightarrow PSM \rightarrow \mathit{code}$ approach where in place of simple transformations we can use chains of transformations of the form $PIM \rightarrow PIM_1 \rightarrow \ldots \rightarrow PIM_n \rightarrow PSM \rightarrow PSM_1 \rightarrow \ldots \rightarrow PSM_m \rightarrow \mathit{code}$. Or we can use a single transformation to merge many input models (see figure 4.5b); but the most important thing to consider is that models and meta models can be anything, from the design project of a system, to the representation of a data library, to some kind of software quality evaluation model; we are not bound to the canonical meaning given to PIM and PSM models. Indeed we can identify two different kinds of transformations:

- *vertical* transformations: the transformation is a refinement of the input model but we are always into the same domain. An example is the $PIM \rightarrow PSM \rightarrow \mathit{code}$ transformation where we are always into the functional domain.

- *horizontal* transformations: the transformation of the input model leads to a completely different domain. An example is the topic of this thesis where we go from a functional domain to a performance domain.

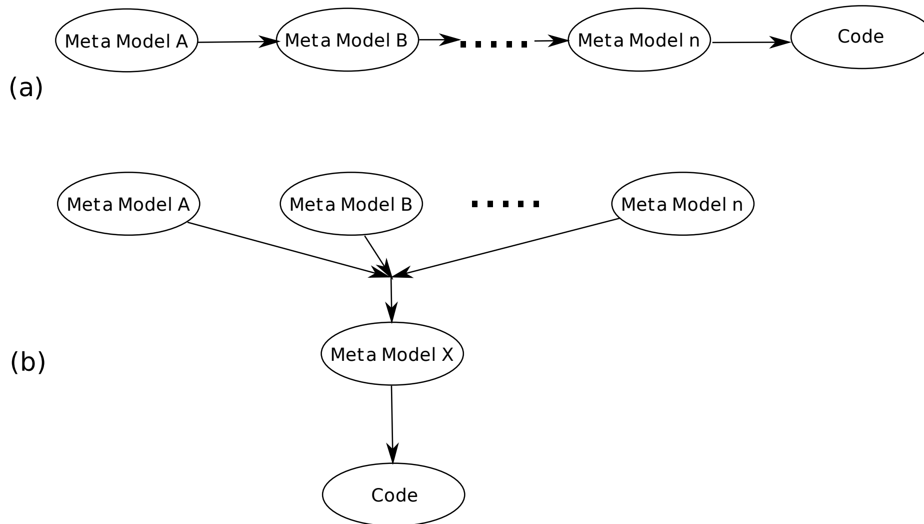

Figure 4.5. (a) Transformation chain. (b) Models merging transformation.

4.3 MDA today

MDA is not an easy topic and therefore it is perceived in many different ways. It is strongly diffused into the academic world with a lot of universities and research centers focused on many aspects of its applications, from code generation to design models, from software quality prediction to Domain Specific Languages and so on.

The industry world on the contrary is not so addicted to MDA methodologies, due to its traditional fear to embrace risks of new technologies even if MDA is around from more then ten years (but the tools that make it possible to effectively use MDA are very young yet). Except for some specific realities (like those born specifically to advise model driven[1]) the adoption of model driven approaches is still very limited even if in the last years its popularity is gaining more and more fans.

4.3.1 Applications

But how does people use MDA today? In the industry world MDA is mainly used to automatically generate code and documentation from design models expressed in UML (remember the $PIM \rightarrow PSM \rightarrow code$ paradigm presented in the previous

[1]For example in Germany we have Itemis AG and in France we have Obeo

section); in this kind of environment usually there are no true M2M transformations. If there are some, they are only used to apply some specific patterns to the input model in order to reduce manual operations of designers. Another field of application is that of Domain Specific Languages where using MDA we want to automatically generate ad-hoc parsers to convert some text to a model and then transform this model to a new one and generate code from the last model.

If instead we consider the research world we can see that MDA is used in many different ways and typically not to generate code like C, Java or C++. On the contrary, the aim usually is to build transformation chains ranging from simple ones to very long and complex transformations where the source and the target model are very far from each other.

Our work lays at an intermediate point between all that. As we will see later our source model will be something that is produced during the design phase (like an UML model for example) whereas our target model will be a software quality evaluation model, but we will not use a single transformation to obtain such a result.

4.3.2 Tools

Even if MDA is not recent from a conceptual viewpoint, the relevance of MDA has grown only recently. The reason is that only recently some working tools have been made available, tools that really allow to follow model driven approaches. In this section we will outline some of them.

Rhapsody

Rhapsody is a design environment developed by Telelogic (now part of Rational Software and so of IBM). It has an integrated environment for developing systems using UML and recently to use SysML; from these models it is able to generate code for some programming languages. The problem with this tool is that despite it claims to be an MDA tool it has really nothing to do with MDA, it only allows to make some very simple customizations to the code generator, but nothing to do with M2M or M2T transformations and therefore with MOF and MDA.

Artisan Studio

Studio (see [52]) is a tool developed by Artisan Software. It is a very good tool for UML and SysML design modeling. It doesn't offer any MOF implementation and the support to M2M transformations is very limited, but M2T transformations are supported even if the transformation language is very distant from the OMG specification given for QVT. Artisan Studio actually can be considered the model driven tool that requires less specific knowledge of MDA if compared to all the other tools.

Eclipse

Eclipse (see [10]) is a very particular instrument because it is not a simple tool, rather it is an ecosystem of tools integrated between them. In such an ecosystem of tools (or plugins to speak in the Eclipse way) there are a lot of instruments that combined together can give the best MDA tool available today with regard to completeness and versatility. The drawback of Eclipse is that it requires a very deep knowledge of model driven approaches and of all the technologies related to MDA.

TopCased

TopCased is a project sponsored by many industrial partners and by the European commission with the VII Framework program (see [1]). It is mainly focused on embedded systems but it can be used in any software domain without any problem. The project is structured over Eclipse and in effect it is capable of using all the potentiality of the Eclipse ecosystem. We have considered it separately because it provides one possible and mature way to use some of the instruments already present into Eclipse. From a user point of view TopCased has some nice UML and SysML editors, a documentation generator, one implementation of QVT for M2M transformations and another one for M2T transformations and also it provides some tools specifically focused to embedded applications such as a simulator for state machines. Its power is obviously the underlying Eclipse environment that makes even access to MOF.

4.3.3 The Eclipse world

Eclipse deserves more attention than other tools because actually it is the only instrument that, despite the significant learning curve, is capable to offer all the instruments needed to follow a true model driven approach; we can say that a considerable part of the success of MDA of these last years is due precisely to Eclipse.

Eclipse is the first tool that provides a really usable implementation of MOF called EMF (Eclipse Modeling Framework) developed inside the Modeling subproject of Eclipse (see [32]).

Quite a number of tools has been built around EMF. Here we will describe the most successful ones related to MDA and in particular to models transformations:

- **GMF** (see [13]): is a framework to build graphical environments (usually graphical editors) based on a meta model.

- **ATL** (see [12]): it is a project from INRIA to implement M2M transformations; in effect it was the first project for M2M and therefore it is the most distant from the actual QVT standard, but due to its long history it is widely adopted.

- **xTend** (see [18]): it is the M2M language from Itemis AG born from the openArchitectureWare project. It is more close to the QVT standard than ATL and has a lot of OCL functionalities of QVT, but it is not yet a full QVT implementation. If needed can be extended with Java classes.

- **QVTO** (see [15]): QVT Operational is an Eclipse implementation of QVT in its operational form. It complies with the QVT specification but it is still quite young, even if it is very promising.

- **QVTR** (see [16]: QVT Relational is an Eclipse implementation of QVT declarative. Actually it is not yet suitable for a real use.

- **JET** (see [14]): Java Emitter Templates is the most basic tool for M2T transformations. It generates code starting from templates based on a syntax very similar to JSP.

- **Acceleo** (see [11]): it is another code generator based on templates, but if compared with JET it is more powerful and the scripting language offers more functionalities. It is even more simple to use.

- **xPand** (see [17]): yet another template language for M2T transformations; conceptually it is very close to Accelleo, but it may rely on a full integration with xTend and its java extensions.

In addition to these sub-projects EMF ads a number of very useful tools like models and meta models converters, models and meta models diff and merge and so on.

Eclipse is a really exciting and full of life project and actually it is one of the most important driving force of the MDA environment. Actually all the work of this thesis is developed under Eclipse.

Chapter 5

The KLAPER approach: basic concepts

5.1 The intermediate meta model concept applied to performance and reliability analysis

In chapter 3 we saw some of the models typically used for software quality analysis, but we also saw that they are very far from what typically is produced during a usual software design phase. To tackle this problem we presented in chapter 4 a development methodology whose main feature is the use of a model based approach specifically focused on models definition and models transformations.

Working with models we are able to fill the gap between design artifacts and software quality evaluation techniques. But we can follow many ways to achieve our goal.

One possible approach could be to define a specific transformation from one design meta model (because transformation rules refer to meta models and then they are applied to models) to a specific software quality analysis meta model; then if for some reason we have to add another quality meta model (different from the previous one) we have to add another specific transformation to this new meta model. But what happens if we introduce a new design model to use as input for models transformations? With the use of MDA we just proposed we should define two new transformations: one from the second input design meta model to the first quality meta model and one from the same input design meta model to the second

quality meta model. It is simple to understand that in this way the number of needed transformations dramatically increases each time a new meta model is added (see figure 5.1 for a graphical representation of this concept).

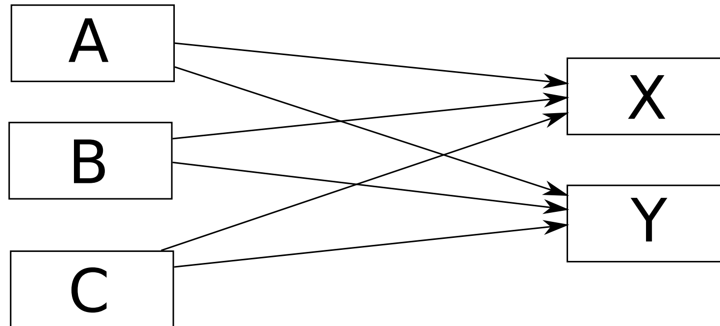

Figure 5.1. Many to many MDA transformations

We think that this is not the best possible approach to solve the problem because it requires too much effort to add the needed transformations each time a new meta model is introduced (either a design meta model or a software quality analysis one).

A better approach to the problem is to use an intermediate meta model to reduce the number of needed transformations.

In chapter 4 we said that transformations can be composed into a chain; but we have to understand that using a specific transformation or a chain of transformations can lead to the same result if the source meta models are the same and they are transformed into the same final target meta model, or at least this is true if the transformations chain applies an overall transformation identical to the single direct transformation. Consequently we can introduce into our approach an intermediate meta model that can be used to reduce the number of needed transformation as shown in figure 5.2. With this kind of approach, every time a new meta model is added we have to write only one new transformation from the new meta model to the intermediate meta model if the new one is a design meta model, or from the intermediate meta model to the new meta model if the new one is a software quality analysis meta model.

In this chapter we present a possible implementation of the concept of the intermediate meta model to fill the gap between design models and software quality evaluation models. Its name is *KLAPER*.

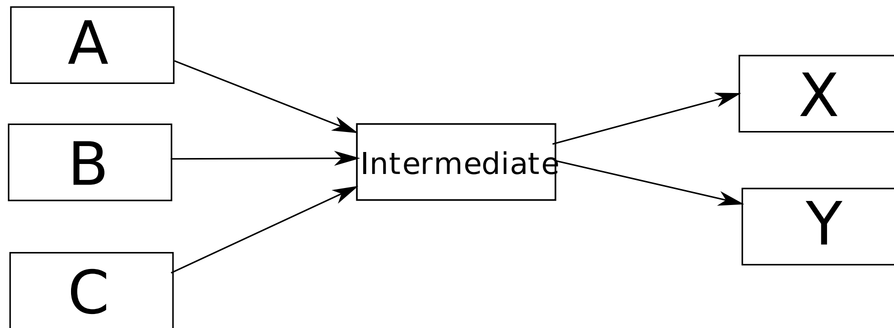

Figure 5.2. Use of an intermediate meta model to reduce the number of transformations

5.2 The meta model

KLAPER (Kernel LAnguage for PErformance and Reliability analysis, see [47] and [26]) is an intermediate language whose aim is to simplify the transformation process between design and software quality analysis models. It is mainly focused on performance and reliability analysis and as an “intermediate” language it contains some typical elements of design models and some typical element of performance and reliability evaluation models.

5.2.1 An overview

A KLAPER model consists of two parts: a static one and a dynamic one. The static part describes the structure of the system from the (hardware and software) available resources and offered services point of view. The dynamic part instead describes the way resources interact between them using services.

Meta model static part: the structure

The static part of KLAPER describes the resources of the system and the services offered by each resource.

Inside a KLAPER model (see figure 6.3 for the KLAPER meta model) we can find a set of *Workload* and a set of *Resource*. The former represents the way the internal or the external workload (usually human users or other systems) interacts with the system. The latter describes the resources of the system, where with

the word "resource" we refer to any component, both hardware and software, of the system that represents an entity capable of offering one or more services. Each resource provides some *Service* that define the services offered by that resource, that is actions that resource provides to everyone will require them; requested services can rely on parameters expressed by *FormalParam*.

Meta model dynamic part: the behavior

The dynamic part of KLAPER describes the sequences of actions accomplished by every system service in terms of a graph of activities and interactions with other services; that means the the dynamic part of KLAPER describes the behavior of the system.

Each service is defined by a *Behavior* which describes the dynamic behavior realized by that service; in other words the Behavior specify the *Step* sequence executed by a particular service and the way these Steps are linked together by some *Transition* to realize the provided services (Transition are designed to define the Step execution sequence).

Steps can be simple or composed and define actions executed inside the service. Simple steps can be of different types: *Start* defines the beginning of an actions flow, *Wait* handle the wait for the receipt of a specific event, *End* describes the end of an actions flow. Are part of simple steps also *Control* steps that are in charge to describe the actions flow inside a Behavior. Other simple steps are: *Branch* that handles decision branches, *Fork* that creates parallel execution flows and *Join* that synchronizes the parallel execution flows previously generated by one or more Fork.

A very important step is *Activity*. In its simplest form Activity (called "simple Activity") describes an atomic activity (often a single action) executed by the system. If considered in its more complex form Activity (called "complex Activity") describes an entire Behavior (in this case considered as a sub-Behavior) as the action to do. A complex Activity can be used to describe loops inside Behaviors' action flow.

An extension of Activity is the *ServiceControl* step; it represents a service request to another resource (or even to the same resource) realizing the interaction between resources of the same system; this interaction can be parametric and in that case parameters have to be specified using *ActualParam* (mirror to FormalParam).

As just said ServiceControl allow a Behavior to request services, but to do that we need a *Binding* that links the service that must be executed. The Binding concept is probably one of the most important elements of KLAPER because it relies on this element for the capacity of dynamically reconfiguring system components according to variations that can occur for design decisions (for example the deployment of the system is changed to improve performance or a component is replaced with another one) or for a dynamic runtime reconfiguration (for example an adaptive client/server system capable of self reconfiguration to balance servers load). The Binding main role is to define the specific service called by ServiceControl, suggesting whether the interaction with this service is synchronous (and therefore we are speaking about a true message between services) or asynchronous (and therefore we are speaking of an event).

Dynamic system reconfigurations at runtime are done thanks to two specific steps called *CreateBinding* and *DeleteBinding*; from the names we can understand that the former is used to create a new Binding while the latter has the role to destroy an existent Binding.

KLAPER: what to map and where

Now that we know a little bit more how the KLAPER meta model is structured, we can see some example of what is mapped into the main KLAPER meta classes from the design world. Just to give some example:

- Each software component (expressed for example using SysML or UML notation) is mapped to a KLAPER Resource. The services offered and required by each software component are mapped to KLAPER Services contained into the related Resources.
- Hardware components supporting the execution of software components of the system are mapped to KLAPER Resource (just like software components).
- The sequence of actions performed by an actor of the system and usually expressed for example with an UML Sequence diagram is mapped to the KLAPER Workload meta class. Usually to complete the mapping, the information provided by the pure UML is not enough and therefore UML is augmented with some additional notation (typically UML-SPT [43] or MARTE

[42]) whose aim is to express concepts like think time, inter-arrival rate, system population and so on.

- The sequence of actions of a service (for example consider a UML sequence diagram or a UML activity diagram) is expressed using a KLAPER Behavior.
- Typical control constructs like *loops*, *if*, *for* and so on are mapped to the related KLAPER control Step.
- Function calls, method calls and more in general the request for a service issued by some system resource (both hardware and software) are mapped to KLAPER ServiceControl step. This step is then associated to a suitable Service offered by some Resource through a Binding. We point out that the modification of Bindings is the way used in KLAPER to model the dynamic reconfiguration of the system at runtime when needed.
- The execution of a software component over an hardware component is modeled as follows: the software component is mapped to a KLAPER Resource with a certain Service (but the Resource can have more Services), the hardware component is mapped to a KLAPER Resource with a single Service that represents the basic function of the hardware component (for example for a Resource that models a cpu we can have a single Service called "process"), the Resource representing the software component is linked to the Resource representing the hardware component using a Binding between the related Services. We have to note that Bindings are used to model two different situations: the interaction between two software components (discussed in the previous point) and the deployment of a software component over an hardware node (discussed in this point).
- Actions executed into a service and expressed for example using the body (code) of a method of a UML class, are mapped to KLAPER Activity; here we usually find also some additional informations related to performance, like service times, and reliability, like failure times of failure probabilities, that are directly mapped into KLAPER Activities' attributes.
- Some quantities used in the design model, like the number of bytes transfered over a network or the number of basic operations needed to complete an action,

are mapped to KLAPER FormalParam and ActualParam.

- Handling the acquisition and release of finite capacity resources is mapped to the *Acquire* and *Release* steps where the acquired or released Resource is specified.

These are only few examples of how some concepts typical of the design world are mapped to the KLAPER related entities; we have not the claim to fully explain here all the aspects of the mapping from some design models (like UML for example) to KLAPER; we just wanted to give an idea of how concepts from the design model can be mapped to corresponding concepts of KLAPER, expressed by its meta classes. We refer to section 10.2.1 for a complete example of the mapping from a design model (in the specific case UML) to KLAPER.

For a better and detailed explanation of each KLAPER meta class with the related attribute see the section 6.1.3

KLAPER for performance and reliability

The KLAPER meta model (the static part plus the dynamic one) is based on the concepts of resources and services that perform activities and interact among them; however, KLAPER does not model the semantics of these services and interactions (i.e. the function they perform). KLAPER is just interested in modeling how long it takes to complete an operation, or if it can fail before completing. This is only one of the possible ways to consider a system. The advantage of such a view is that we can concentrate on the structure of the system and on the interaction between its components without any knowledge of the expected functions of the system itself. But now that we are able to model a system independently of its functional requirements, how can we use it to analyze performance and reliability? To answer to this question we need to recall that KLAPER is an "intermediate" language created to link two worlds: the software design one and the software analysis one.

From the software design world KLAPER takes an abstracted view of all the elements needed to represent the static structure (the static part of KLAPER) and the dynamic behavior (the dynamic part of KLAPER) of the system we want to analyze; then from the software analysis world KLAPER takes a number of concepts used to enrich the static and the dynamic parts just presented. Indeed concepts like activities or service call, typical of design models, are augmented with informations

like service time or failure probabilities that are instead typical of quality analysis models.

More specifically the fundamental concepts allowing performance and reliability analysis are:

- The execution time needed by an activity to be fully completed.
- The failure probability of an activity.
- The time spent by an activity before a failure.
- The dependence of an activity from a called service in terms of failure.

These concepts (all represented using attributes of KLAPER Steps) applied to the static and dynamic description of the system given by the KLAPER meta model, are what we need to successfully complete a performance and reliability analysis of a software system.

One more thing very important to note is that all the concepts just described (like for example the failure probabilities, the service times but also the repetitions of activities) and used to analyze the performance and the reliability of the input system are expressed using probability distribution functions. That means that all the KLAPER approach relies on a "stochastic vision" of the analyzed system.

Chapter 6

KLAPER: syntax and semantics

6.1 Meta model packages

The KLAPER meta model consists of three packages: core, probability and expr. In the next sections we will analyze each of these packages.

6.1.1 The probability package

Into the probability package, shown in figure 6.1, we have all the probability distribution functions that we can use in KLAPER with theirs attributes:

- Normal distribution function. Its attributes are reported in table 6.1.

Attribute	Type	Description
mean	Expression	mean of the distribution.
standDev	Expression	standard deviation of the distribution.

Table 6.1. Normal distribution function attributes

- Poisson distribution function. Its attributes are reported in table 6.2.

Attribute	Type	Description
mean	Expression	mean of the distribution.

Table 6.2. Poisson distribution function attributes

- Uniform distribution function. Its attributes are reported in table 6.3.

Attribute	Type	Description
min	Expression	minimum value of the distribution.
max	Expression	maximum value of the distribution.

Table 6.3. Uniform distribution function attributes

- Exponential distribution function. Its attributes are reported in table 6.4.

Attribute	Type	Description
mean	Expression	mean of the distribution.

Table 6.4. Exponential distribution function attributes

- Constant: it represents a constant value and therefore is not properly a probability distribution function but defined in this way is only a workaround to simplify the meta model structure. Its attributes are reported in table 6.5.

Attribute	Type	Description
value	Expression	expression of the constant value.

Table 6.5. Constant

- Geometric distribution function. Its attributes are reported in table 6.6.

Attribute	Type	Description
mean	Expression	mean of the distribution.

Table 6.6. Geometric distribution function attributes

- Histogram distribution function. Its attributes are reported in table 6.7.

We must pay particular attention to the Constant meta class that is used to represent a general constant value even if it derives from the meta class ProbabilityDistributionFunction which really has nothing in common, but doing so the meta model is simpler to build.

Attribute	Type	Description
samples	HistogramSample	sample of the histogram.

Table 6.7. Histogram distribution function attributes

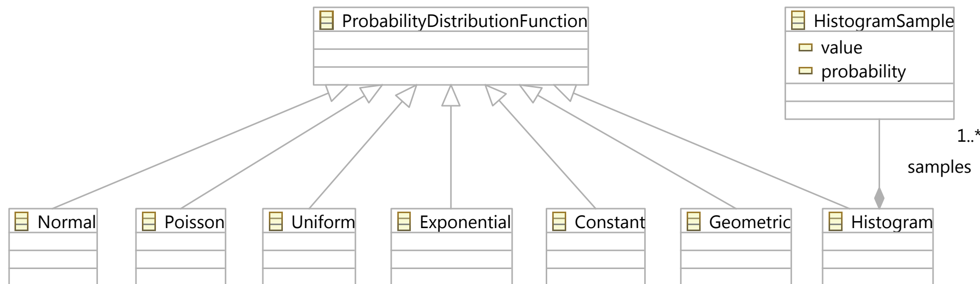

Figure 6.1. Package probability of KLAPER meta model

6.1.2 The expr package

The expr package (see figure 6.2) contains a meta model that represents expressions formed by constant values, variables, unary operators and binary operators. The meta classes composing this package are:

- Expression: is the root of an expression; it has no attributes.
- Atom: represents an atomic element like for example a number or a variable; it has no attributes.
- Variable: meta class that represents the concept of a simple variable (for example into the expression $x + y$, x and y are variables); it has no attributes.
- Number: is the general concept for any kind of number; it has no attributes.
- Integer: meta class that represents a number contained into the $\mathbb{Z}$ set; its attributes are reported in table 6.8.
- Double: like Integer is a meta class that represents a number but contained into the $\mathbb{R}$ set; its attributes are reported in table 6.9.

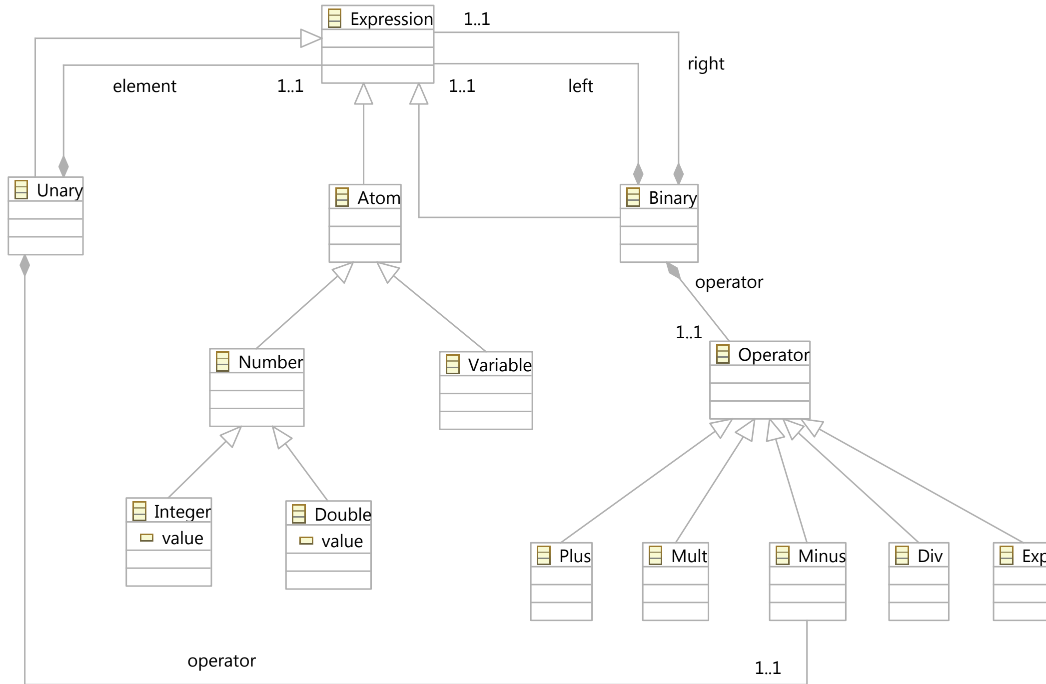

Figure 6.2. Package expr for the KLAPER meta model

Attribute	Type	Description
value	Integer	value is a number that $\{value \mid value \in \mathbb{Z}\}$

Table 6.8. Integer attributes

Attribute	Type	Description
value	Integer	value is a number that $\{value \mid value \in \mathbb{R}\}$

Table 6.9. Double attributes

- Unary: meta class that represents an element composed only by a single sub-expression without any binary operator. The only unary operator defined is the minus symbol; its attributes are reported in table 6.10.

- Binary: meta class that defines a binary operator; its attributes are reported in table 6.11.

Attribute	Type	Description
operator	Minus	the only unary operator admitted is the minus symbol.
element	Expression	a recursive expression.

Table 6.10. Unary attributes

Attribute	Type	Description
operator	Operator	the binary operator (admitted binary operators are +, -, *, / and ∧).
left	Expression	the recursive expression on the left of the operator.
right	Expression	the recursive expression on the right of the operator.

Table 6.11. Binary attributes

- Operator: represents a binary operator; it has no attributes but has five derived meta classes: Plus (for the binary operator "+"), Minus (for the binary operator "-"), Mult for the binary operator "*"), Div for the binary operator "/") and Exp (for the binary operator "∧").

With this kind of meta model we can model each expression, even the most complex one.

6.1.3 The core package

The core package is the true heart of the KLAPER meta model (probability and expr are only utility packages). In section 5.2.1 we already saw an overview of the meta classes that compose the core package; here we will describe the details of each of these classes:

- **KlaperModel**: is the container of all Workload and Resource composing the system model; it has no attributes.
- **Resource**: a meta class that represents any hardware or software entity able to provide a service; its attributes are reported in table 6.12.
- **Service**: this meta class describes the service offered by a resource; its attributes are reported in table 6.13.

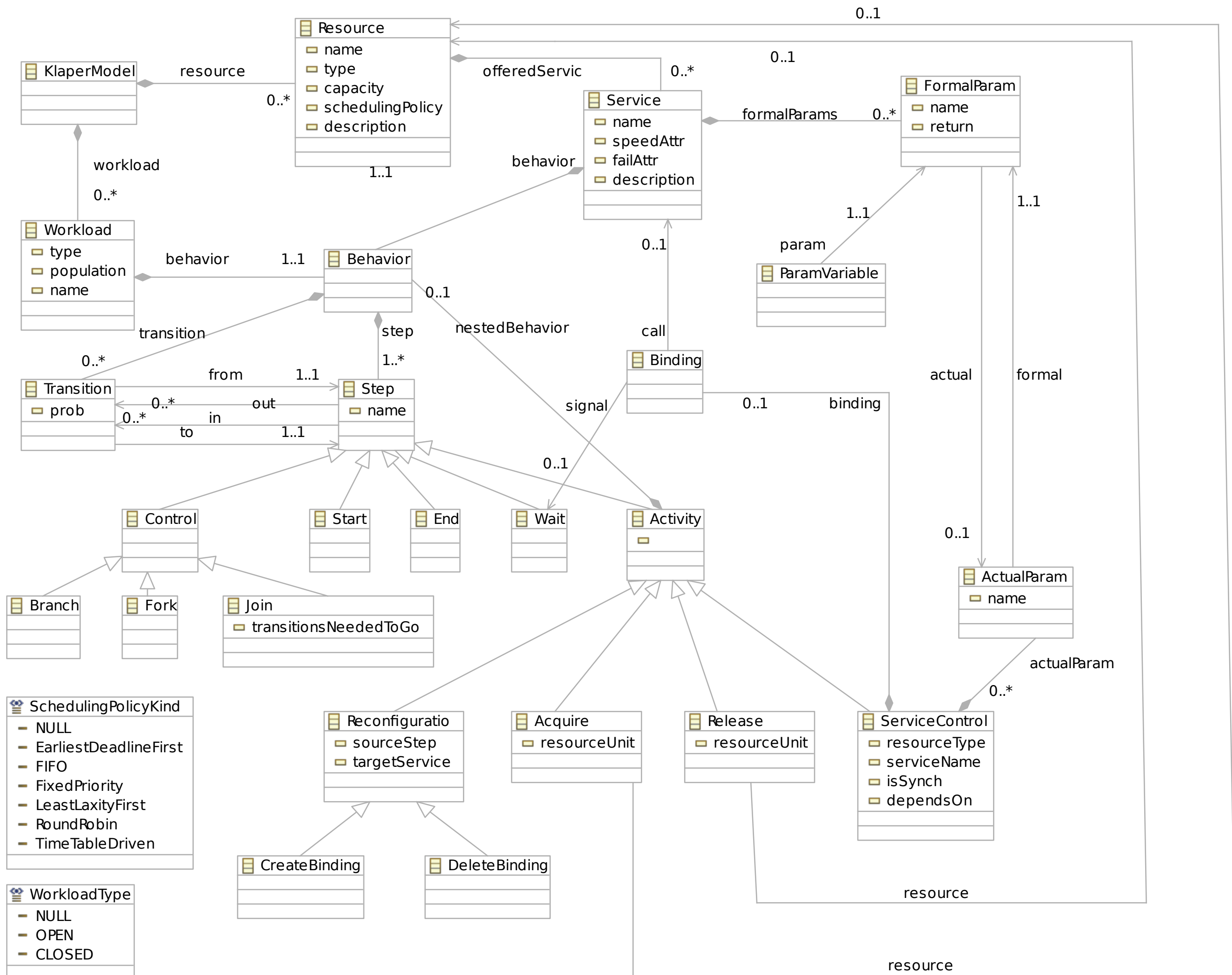

Figure 6.3. Package core of KLAPER meta model

- **FormalParam**: used to parameterize a service offered by a resource (the concept is very similar to that of parameters in many programming languages, but this time parameters refers to performance and reliability aspects); its attribute are reported in table 6.14. FormalParameters can be input, output

[1]At the time this work is written we are evaluating if speedAttr and failAttr concepts can be merged with Service's FromalParams. The discussion is about what, if any, can be represented with failAttr and speedAttr that can't be represented with FormalParams

Attribute	Type	Description
name	String	the name of the resource.
type	String	type of the resource (some types like “cpu”, “network”, “disk” are managed in a special way).
capacity	Real	denotes the number of requests served in parallel by the resource. It represents the overall capacity of the resource considering all the requests to all the services offered by the resource.
schedulingPolicy	Enum	requests scheduling policy applied by the resource. There is the same policy for al l the services offered by the resource.
description	String	a text description of the resource.
offeredService	List	a list of services provided by the resource.

Table 6.12. Resource attributes

Attribute	Type	Description
name	String	the name of the service.
speedAttr[1]	Real	is used to represent the basic parameters characterizing the resource “speed” in carrying out the service; in the simplest case it could be a renaming of the Real type, expressing for example the number of operations per unit time for this service. In more complex cases it could specify additional informations such as the time scale used in the definition of this parameter.
failAttr[1]	Real	. the same as speedAttr but related to the internal failure probability of the service.
description	String	text description of the service.
behavior	Behavior	sequence of actions (steps) that realize the service.
formalParams	List	FormalParams of the service. They can be used to parameterize expressions concerning the time to complete an operation or the probability that it fails from the point of view of the performance and reliability analysis.

Table 6.13. Service attributes

or input/output parameters. Input parameters are used when the called service depends on some external variable; output parameters are used when we

need the result of a service execution to compute some other parameter or expression (for example we can consider a service that computes the number of elements contained into a list, we can call this service and later we can use the returned size of the list to compute the value of some expression like for example the processing power needed to sort the list), just like reference or pointer parameters of common programming languages; input/output parameters are used when we want to use parameters both in input and output mode.

Attribute	Type	Description
name	String	name of the parameter.
return	Boolean	true if the parameter is an output or an input/output parameter, false if it is only an input parameter. Output parameters and input/output parameters can be used when the service time or the failure probability of an Activity of a Service depend on some other Service previously called.
actual	ActualParam	the value that actualize this parameter.

Table 6.14. FormalParam attributes

- **Workload**: meta class that describes the internal or external system workload; it is a particular form of service that doesn't provide any service but only requires services from other resources; its attributes are reported in table 6.15.

- **Behavior**: meta class that describes a sequence (or better a direct graph) of steps or actions needed to provide a specific service; it attributes are reported in the table 6.16.
- **Step**: is the basic action into a Behavior; there are two types of steps: simple steps and complex steps; its attributes are reported in table 6.17.
- **Transition**: meta class that represents the transition from a step to another one; its attributes are reported in table 6.18.

[2] An open workload is characterized by users that enter the system, do some requests and then go away

[3] A closed workload is characterized by a constant number of users that when a request is served do another request

Attribute	Type	Description
name	String	name of the workload.
type	Enum	the type of the workload; admitted types are OPEN for an open workload[2] and CLOSED for a closed workload[3].
population	Integer	the number of users inside the system for a closed workload (in mutual exclusion with arrivalProcess attribute).
arrivalProcess	ProbabilityDistributionFunction	distribution function of user arrival rate for an open workload system (in mutual exclusion with population attribute).
behavior	Behavior	sequence of actions (steps) that realize the workload.

Table 6.15. Workload attributes

Attribute	Type	Description
step	List	a list of steps that build the behavior (they compose a direct graph).
transition	List	the links between the steps.

Table 6.16. Behavior attributes

Attribute	Type	Description
name	String	name of the step.
in	List	list of incoming transitions into the step.
out	List	list of outgoing transitions from the step.

Table 6.17. Step attributes

- **Start**: it is the starting step of each Behavior and is mandatory. It is a simple step. Its attributes are reported in table 6.19.

Attribute	Type	Description
from	Step	source step of the transition.
to	Step	target step of the transition.
prob	Real	the probability that the Transition is chosen where more than one Transition is available; if not defined defaults to the value 1.

Table 6.18. Transition attributes

Attribute	Type	Description
name	String	see Step.
in	List	empty list.
out	List	see Step but the list can have only one element.

Table 6.19. Start attributes

- **Wait**: it is a step that freezes the execution flow of a Behavior until the waited event is raised[4]. It is a simple step. Its attributes are reported in table 6.20.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.

Table 6.20. Wait attributes

- **End**: it is the final step of each Behavior and it is mandatory. It is a simple step. Its attributes are reported in table 6.21.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	empty list.

Table 6.21. End attributes

[4]The waited event is raised when it is executed the ServiceControl that is linked to the Wait step via a Binding element.

- **Control**: this meta class is a specific specialization of Step that models steps used to control the execution flow of a Behavior. It is a simple step. Its attributes are reported in table 6.22.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.

Table 6.22. Control attributes

- **Branch**: its a control step that model a branching point into the execution flow of a Behavior. It is a simple step. Its attributes are reported in table 6.23. The sum of the values of the *prob* attributes of this step's outgoing Transitions must be equal to 1.0.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step.

Table 6.23. Branch attributes

- **Fork**: it is a control step that causes the creation of two or more parallel execution flows into the same Behavior. It is a simple step. Its attributes are reported in table 6.24.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step.

Table 6.24. Fork attributes

- **Join**: it is a control step where parallel flows originated from a Fork step join together. It is a simple step. Its attributes are reported in table 6.25.

The *transitionsNeededToGo* attribute is used to define the number of flows that we need to wait before going to the next step; for example if its value is set to 1 that means that the step has to wait to be activeted by one only incoming transition, if its value is set to 3 we have to wait at least three incoming transitions before going to the next step. In general if we have n incoming transitions and the attribute *transitionsNeededToGo* is set to 1 the continuation condition can be considered as the OR between the predicates that represent the termination of the activities that merge into the Join step, while if the attribute is set to n the continuation condition can be considered as the AND between those predicates; obviously all the intermediate values between 1 and n are accepted (the value 0 on the contrary is not acceptable).

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
transitionsNeededToGo	Integer	number of transitions to gather before proceeding.

Table 6.25. Join attributes

- **Activity**: meta class that represents an action into a Behavior. There are two types of Activity: simple (to describe atomic actions) and complex (to describe actions composed by some other activities specified by a sub-Behavior). Its attributes are reported in table 6.26.

  In table 6.26 we said that loops can be modeled using the *repetitions* attribute. To better explain this concept in figure 6.4 we present an example of how a loop with a repetitions number greater than 1 can be modeled using *repetitions* and how instead the same loop can be represented in KLAPER not using *repetitions*; the two representations are equivalent but obviously the first one must be preferred due to its simplicity. In the example we have a repetitions number represented by a constant value, but we can also express the number of loops using a stochastic (parametric) distribution.

- **ServiceControl**: it is a special activity that makes calls to services of other

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
repetitions	ProbabilityDistributionFunction	times the activity is executed before going to next steps (this can be used to model loops, for an example see figure 6.4).
nestedBehavior	Behavior	defines a complete Behavior as the action to do with this step.
internalExecTime	ProbabilityDistrbutionFunction	time required by the system to complete the action related to this step. Can depend on Service's FormalParams.
internalFailProb	ProbabilityDistributionFunction	the probability that the activity will fail; it is used for reliability analisys. Can depend on Service's FormalParams. It is in mutual exclusion with internalFailTime
internalFailTime	ProbabilityDistributionFunction	the time before a failure will happen during the activity execution; it is used for reliability analisys. Can depend on Service's FormalParams. It is in mutual exclusion with internalFailProb.

Table 6.26. Activity attributes

resources or of the same resource or raises events. Its attributes are reported in table 6.27. Special attention is required by the *dependsOn* attribute because it plays a very important role during reliability analisys. In ServiceControls

Attribute	Type	Description
name	String	see Activity.
in	List	see Activity.
out	List	see Activity.
repetitions	ProbabilityDistributionFunction	see Activity.
nestedBehavior	Behavior	see Activity.
internalExecTime	ProbabilityDistrbutionFunction	see Activity.
internalFailProb	ProbabilityDistributionFunction	see Activity.
InternalFailTime	ProbabilityDistributionFunction	see Activity.
resourceType	String	the type of the resource (for example "cpu", "network", etc.) that would provide the required service. If the deployment of the system is already available this information is useless, but if the deployment has not been defined yet this information could be useful for some type of analysis.
serviceName	String	the name of the required Service.
isSynch	Boolean	true if the service request is synchronous (and therefore blocking), false if it is asynchronous (in that case the requester doesn't block waiting the response from the requested service). If ServiceControl is used to raise a signal this attribute must be false.
binding	Binding	the Binding linked to this ServiceControl. If the deployment of the system is not yet available, this reference can be empty.
ActualParam	List	a list of actual parameters with which the object is invoked.
dependsOn	Boolean	true if a failure of the called service causes a failure also into this step, false if a failure of the called service is not a problem for this step. Used to evaluate reliability.

Table 6.27. ServiceControl attributes

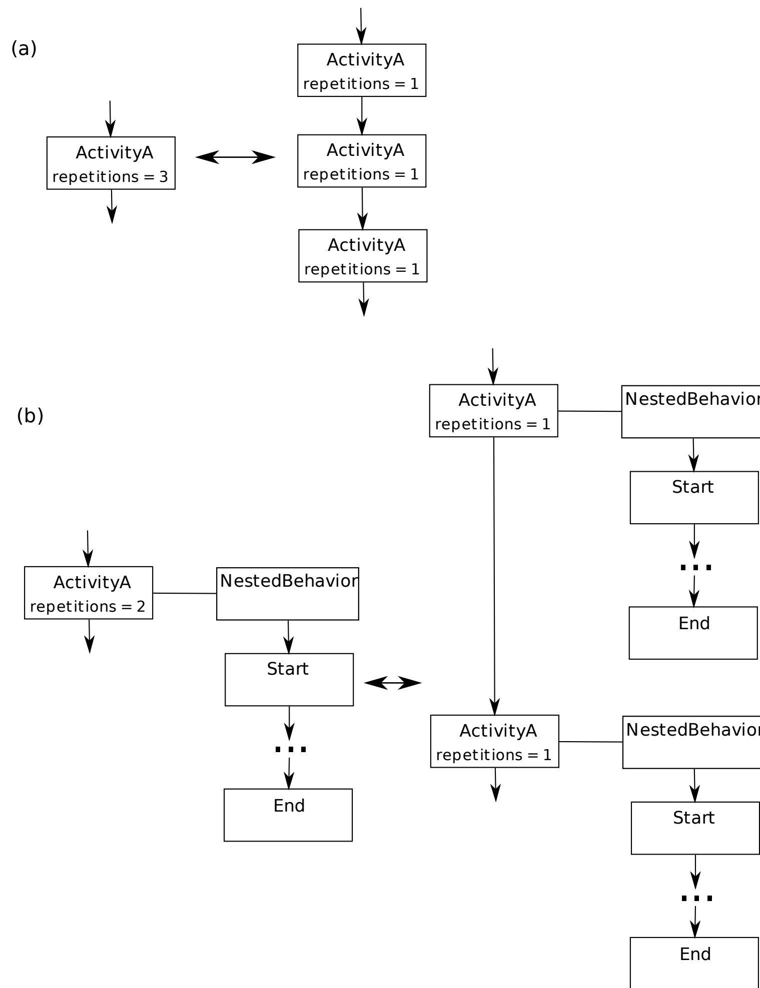

Figure 6.4. (a) Loop unfolding for a simple KLAPER Activity, (b) Loop unfolding for a KLAPER Activity with a nested behavior.

with a synchronous call the calling service is obviously dependent on the called service because the caller is blocked waiting the called to complete its activities; therefore a failure of the called service must cause also a failure of the caller service; but when we have a ServiceControl with an asynchronous call to another service this dependency of failures is not so obvious, there are cases where a failure of the called service causes a failure of the caller service and cases where this does not happen; the *dependsOn* attribute aim is precisely to indicate whether or not a ServiceControl with an asynchronous call must fail

if the called service has a failure.

- **ActualParam**: this meta class represents the real value of a formal parameter assigned when a service is invoked by a ServiceCall. Its attributes are reported in table 6.28.

Attribute	Type	Description
name	String	name of the parameter.
formal	FormalParam	the formal parameter which this parameter assigns a value (for an example see figure 6.5).
value	Expression	the expression (see the expr package) that defines the value of this parameter.

Table 6.28. ActualParam attributes

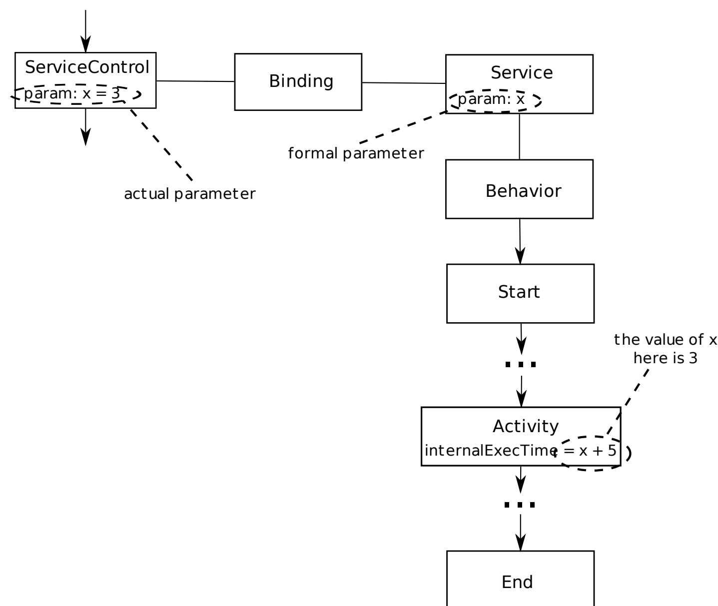

Figure 6.5. Example of correspondence between KLAPER ActualParam and FormalParam.

- **Binding**: meta class that defines the link between a calling service (using a ServiceCall) and a called service and defines if the call is a message or an event/signal. Its attributes are reported in table 6.29.

Attribute	Type	Description
call	Service	the service requested with the message.
signal	Wait	see the Wait steps this event have to wake up.

Table 6.29. Binding attributes

- **Reconfiguration**: it is responsible of reorganizing the links between ServiceCall and Service creating or destroying Bindings as needed.It is a simple Step. Its attributes are defined in table 6.30.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
sourceStep	String	the step object of the reconfiguration.
targetService	String	the service to modify.

Table 6.30. Reconfiguration attributes

- **CreateBinding**: it is a special reconfiguration step with the aim to create a new Binding. It is a simple step. Its attributes are reported in table 6.31.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
sourceStep	String	see Reconfiguration.
targetService	String	the new service to bind.

Table 6.31. CreateBinding attributes

- **DeleteBinding**: it is a special reconfiguration step with the aim to delete an existing Binding. It is a simple step. Its attributes are reported in table 6.32.

- **Acquire**: handles the acquisition of a finite capacity resource. The execution

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
sourceStep	String	see Reconfiguration.
targetService	String	the service to un-bind.

Table 6.32. DeleteBinding attributes

flows is blocked until the available capacity can't satisfy the request. It is a simple step. Its attributes are reported in table 6.33.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
resourceUnit	Integer	units of resource needed (and acquired).
resource	Resource	target resource.

Table 6.33. Acquire attributes

- **Release**: handles the release of a finite capacity resource. It is a simple step. Its attributes are reported in table 6.34.

Attribute	Type	Description
name	String	see Step.
in	List	see Step.
out	List	see Step but the list can have only one element.
resourceUnit	Integer	units of resource released.
resource	Resource	target resource.

Table 6.34. Release attributes

6.2 Meta model semantics

To define the KLAPER meta model semantics we will use Abstract State Machines (ASM) formalism. Even if in [4] ASM are described as a methodology for high level software systems design in opposition to other methods like for example UML, here we will use them to define the semantics of the KLAPER meta model; the choice of such an instrument for the definition of a semantic is due, just as in [6], to the severe rigor and formalism offered by this instrument and especially to the mathematical foundations on top of which ASM are built (see [4]). A semantics defined using ASM is simple to understand and, more important, thanks to its very formal mathematical structure it hardly lends itself to interpretations as instead frequently happens with semantics defined using natural language.

The next sections of this chapter requires at least a basic knowledge of ASM to be correctly understood. To solve this problem we will spend some few words about ASM.

6.2.1 A short ASM introduction

ASM have been mainly developed by E. Borger and F. Stark as an high level design method for complex software systems and can be considered like an extension of the simplest FSM (Finite State Machines).

An ASM is a system consisting of a finite set of rules of the form

$$\textbf{if } \textit{condition} \textbf{ then } \textit{updates} \tag{6.1}$$

where *condition* is a first order formula, without any free variable, whose interpretation can be true or false and is called *guard* because when it happens it involves the application of the linked rule. *updates* is the finite set of function updates of the form

$$f(t_1, \ldots, t_n) = t \tag{6.2}$$

its execution determines the definition or the change (in parallel) of the values of a function; in practice *updates* corresponds to the pair $< location, value >$ and represents the state change unit determined by the change of the location value *location* assuming the value *value*. The concept of *state* inside the ASM corresponds to the concept of abstract data structure, that means data to which it is possible to apply basic operations and predicates (attributes and relations).

More formally (as done in [24]) we can say that an ASM can be defined by the tuple:

$$ASM = \{Header, Body, MainRule, Initialization\} \quad (6.3)$$

where:

- *Header*: contains the name of the ASM and its signature.
- *Body*: consists of function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations of transition rules (also called *update rules*, see 6.1 and 6.2). An ASM M is therefore a finite set of rules; state transitions of M may be influenced in two ways: internally, through the transition rules, or externally, through the modifications of the environment. A *computation* of M is a finite or inifinite sequence S_0,$S_1, \ldots$,$S_n, \ldots$ of states of M where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the update rules which are enabled in S_n.
- *MainRule*: is a transition (or *update*) rule and represents the starting point of the machine program. The main rule is *closed* (that means it doesn't have parameters) and since there are no free global variables in the rule declarations of an ASM , the notion of a move (or state transition) doesn't depend on a variable assignment, but only on the state of the machine.
- *Initialization*: is a characterization of the initial states of the ASM. An initial state defines an initial value for domains and functions declared in the signature of the ASM. Executing an ASM means executing its main rule from a specific initial state.

To understand the behavior of ASM we need to define the concept of *consistency* of an update. A set of updates is defined as consistent if all its updates refer to different locations; in other words if a pair of updates refers to the same location, all the set have to be considered inconsistent.

Defined the concepts of "state" and "rule" for updates we can face how an ASM is executed. ASM are executed by an agent that executes computational steps; a computational step of an ASM in a given state is performing simultaneously all the updates of all the rules whose guard is true in that particular state. The result of the computation causes a state transition only if all the updates are consistent; if there is

an inconsistent pair of updates the computational step does not imply a transition to a new state but we have an error. The ASM agent computation advances in an iterative way going through each computational step of the ASM itself. If the execution of the ASM has an end we can define a specific ending criterion, like for example:

- no more rules are applicable;
- an empty update is executed;
- the state is always the same and doesn't change anymore;
- etc...

An evolution of simple ASM (considered until now) are distributed ASM (see [7] and [28]) where there isn't only one agent that executes updates but there are many agents that execute in parallel each its own sub-ASM. But to have many agents running on a distributed ASM we have to respect some rules (see [7]):

- each update and therefore each state change must have a finite number of predecessors.
- steps (that means updates) of each agent are linearly ordered.
- each state must be the result of the updates of all the previous states (coherence rule).

For simplicity we assume that for the rules just seen, updates and therefore state changes are atomic; in a more general environment they could have considered both atomic and with a period in time.

More detailed information about ASM, their structure and their use can be found in [4] and [3].

6.2.2 Functions for the KLAPER ASM

Functions for meta classes

ASMs rely on functions between sets to represent the *update rules* executed by ASM agents. Therefore the concepts modeled using meta classes in KLAPER have to be

mapped to elements belonging to some set of elements with the related functions needed to retrieve these elements. We need to follow this approach for all KLAPER main concepts like Resources, Services, Behaviors, Steps and so on. In this section we will present all the sets and functions we have introduced to describe the KLAPER meta classes concepts in terms of ASM functions (that later will be used by update rules).

A KLAPER ASM consists of an abstract set RESOURCE containing all the **Resource** entities of the system. Each element of this set has the form

$$resource(name, type, capacity, schedulingPolicy, description, offeredService) \tag{6.4}$$

we can refer to the previous section for the meaning of attributes of this function and of all the other functions that we will see from now on.

With *offeredService* each resource provides some **Service** that are from the abstract set SERVICE and have the form

$$service(name, speedAttr, failAttr, description, formalParams) \tag{6.5}$$

Very similar to Service are **Workload** that are part of the abstract set WORKLOAD. Each Workload has the form

$$workload(name, type, population, arrivalProcess, behavior) \tag{6.6}$$

Services and workload describe the sequence of steps executed with services using the **Behavior** that are part of the abstract set BEHAVIOR. Each Behavior consists of a graph of nodes and arcs respectively called Step and Transition. As seen until now **Transition** are part of the abstract set TRANSITION and have the form

$$transition(from, to, prob) \tag{6.7}$$

while **Step** are part of the abstract set STEP and have the form

$$step(name, in, out) \tag{6.8}$$

But KLAPER has some different types of steps (that means meta classes that extend the meta class Step); all of them are part of the abstract set STEP and have a form very similar to that of Step itself

$$start(name, in, out) \tag{6.9}$$

$$end(name, in, out) \tag{6.10}$$

$$fork(name, in, out) \tag{6.11}$$

and so on, except for the Join meta class that has the form

$$join(name, in, out, transitionsNeededToGo) \tag{6.12}$$

In addition to control step we have also other type of steps like for example **Activity** that has the form

$$activity(name, in, out, repetitions, nestedBehavior, internalExecTime, internalFailProb, internalFailT \tag{6.13}$$

and **ServiceControl** with the form

$$serviceControl(name, in, out, resourceType, serviceName, isSynch, binding, actualParam) \tag{6.14}$$

other steps that derives from Activity are **Acquire** and **Release** that have the form

$$acquire(name, in, out, resourceUnits, resource) \tag{6.15}$$

$$release(name, in, out, resourceUnits, resource) \tag{6.16}$$

the last two steps are **CreateBinding** and **DeleteBinding** that have the form

$$createBinding(name, in, out, sourceStep, targetService) \tag{6.17}$$

$$deleteBinding(name, in, out, sourceStep, targetService) \tag{6.18}$$

ServiceControl meta classes are connected to other services using **Binding**. Bindings are part of the abstract set BINDING and have the form

$$binding(call, signal) \tag{6.19}$$

where *call* is a reference to the Service called when the binding is used as a service call and *signal* is a reference to a Wait step when the binding is used as asignal (in the same sense of an operating system signal for example).

To close this section concerning ASM functions related to KLAPER ASM we have to present the two last meta classes used for parameters management, **FormalParam** and **ActualParam** respectively with the form

$$formalParam(name, return, actual) \tag{6.20}$$

and

$$actualParam(name, formal, value) \tag{6.21}$$

6.2.3 Auxiliary functions

Functions related to KLAPER meta classes are not enough to define the needed update rules; we have to define something more to define KLAPER ASM rules. In this section we will describe some auxiliary functions we created to facilitate the update rules definition later in the next section.

Let us call AGENT the abstract set of agents that execute an ASM. Then, we can define the function

$$\mathit{active} : AGENT \to STEP \tag{6.22}$$

as the function that, given an agent $a \in AGENT$, returns the step it is currently executing with the notation $\mathit{active}(a)$; this is useful to determine where a given agent is within the graph of steps of a Behavior in a given moment. If we consider the unary function *Self* that returns the agent currently in execution, then $\mathit{active}(\mathit{Self})$ returns the step currently executed. For sake of simplicity, we denote it as *active*.

Given a step and a transition connected to it, it is possible to determine the step connected at the other end of the transition by the function:

$$\mathit{opposite} : STEP \times TRANSITION \to STEP \tag{6.23}$$

For example with the function $\mathit{opposite}(\mathit{active}, \mathit{transition})$ it is possible to obtain the step linked to the actual one through the transition *transition*.

To determine whether the agent in execution is actually an agent or a sub-agent (that means an agent that is executing a nestedBehavior), we can use agent's property *isNested* which corresponds to the function:

$$\mathit{isNested} : AGENT \to \{true, false\} \tag{6.24}$$

that returns *true* if the agent is a sub-agent, otherwise it returns *false*.

To determine the Behavior an agent is executing we can use the function:

$$\mathit{currentBehavior} : AGENT \to BEHAVIOR \tag{6.25}$$

For example with *currentBehavior*(*Self*) we get the Behavior of the currently active agent.

If an agent is executing a nestedBehavior, we can get the containing Behavior with the function:

$$outer : BEHAVIOR \rightarrow BEHAVIOR \tag{6.26}$$

An agent during an execution flow can have two different states[1]:

- *running*: when an agent is active and running.
- *suspended*: when an agent is waiting for some event (and thus blocked on a step) that can be the end of a nestedBehavior or of a (synchronous) service call, or even the receipt of an event (see the Wait step).

However it is always possible to obtain or modify (using the binary ASM assignation operator ":=" after the function, see equation 6.28 for an example) the state of an agent using the function:

$$mode : AGENT \rightarrow \{running, suspended\} \cup undef \tag{6.27}$$

where *undef* represents an undefined state. An agent in *undef* state is neither *running* nor *suspended* simply because it is not in execution (for example the agent has completed its execution or has been killed). Changing state from *running* to *suspended* completely blocks the agent execution (it is a blocking operation); when it will be awakened the agent will execute the ASM update rule action immediately next to that of state change.

The action of killing an agent can be done using the macro

$$delete(agent) \equiv mode(agent) := undef \tag{6.28}$$

that simply sets to undefined the agent state removing it from execution.

[1] This is not ASM semantics, this is how we decided to handle ASM agents in our semantics.

When we have to execute a sub-Behavior or when we have to execute parallel execution flows, then to accomplish thise work one or more agents are created; all new agents are created from a parent agent to which they notify their "no longer *running* state" when they complete all their duties. This relationship between agents can be expressed with the function

$$parent : AGENT \rightarrow AGENT \cup \{undef\} \tag{6.29}$$

that returns the parent agent of any agent a or returns the value *undef* if we are trying to get the parent of an agent that hasn't a parent. All the sub-agents of a given agent a are members of the set $SubAgent(a) = \{a' \in \text{AGENT} \mid parent(a') = a\}$.

If we have to know the number of sub-agents already ended generated by a specific agent, then the number of such sub-agents can be given by the function

$$subAgentsCompleted : AGENT \rightarrow \mathbb{N} \tag{6.30}$$

where $\mathbb{N}$ is the set of natural numbers.

To realize loops it is necessary to add a new function that traces the times an agent has consecutively executed a step; we can do that with the function

$$executedFromAgent : STEP \times AGENT \rightarrow \mathbb{N} \tag{6.31}$$

where $\mathbb{N}$ is the set of natural numbers.

If we have to execute a nestedBahvior we have to create a new agent to which we have to assign a step to execute. Since each Behavior must start its execution flow from its Start step (that is mandatory), then we need a function to find such a step

$$start : BEHAVIOR \rightarrow STEP \tag{6.32}$$

that given a Behavior returns its Start step.

When an agent has been created due to a service call then this fact is shown by the function

$$called : AGENT \rightarrow \{true, false\} \tag{6.33}$$

that returns *true* if the agent has been created to serve a ServiceCall, otherwise it returns *false*.

To obtain the Behavior of a service called by a ServiceControl we define the function

$$calledBehavior : STEP \rightarrow BEHAVIOR \cup \{undef\} \quad (6.34)$$

that given a step returns precisely the Behavior of the related service if the subject step is a ServiceControl, while if the subject step is of any other kind the returned value is *undef*.

Transitions have an attribute (*prob*) that represents the probability that a given Transition is chosen when there are more possible transitions available; more precisely all the Transitions outgoing from any KLAPER Step must have the *prob* attribute set to 1.0 (this is true also for Fork steps) to indicate that the outgoing Transition or the outgoing Transitions are always activated; the only exception is the Branch step where the sum of the *prob* attributes of all the outgoing Transitions must be equal to 1.0. To know whether a specific Transition is activated or not we can use the function

$$probabilitySelected : TRANSITION \rightarrow \{true, false\} \quad (6.35)$$

that returns *true* if the input Transition has been actually selected (e.g. by using some random number generator) according to its *prob* probability, otherwise it returns *false*. Obviously this function make sense only for Branch steps; for all other steps it always return true because, as just said, except for the Branch step the *prob* attribute is always equal to 1.0.

Agents can require and release resources. To support these activities we need to know the number of resource units available for a specific kind of resource. This can be known with the function

$$availableResources : RESOURCE \rightarrow \mathbb{N} \quad (6.36)$$

that returns the number of resource currently available (that means not acquired) for a specific kind of resources.

When a resource is not available an agent must wait until someone releases the required units of the required resource; to do that an agent has to set its attributes

requiredResource and *requiredUnits*. When a resource is released by an agent this agent has the responsibility to wake up another agent waiting for the released resources, if any; to find the potentially awaken agents we can use the function

$$\mathit{findAgentRequiring} : RESOURCE \times \mathbb{N} \rightarrow AGENT \tag{6.37}$$

that returns (only) one agent, if any (it can also return an empty set), from those waiting for the availability of the specified resource (the released resource is compared to the awaken agents' *requiredResource* attribute) in a number of elements less than or equal to the resource units released by the agent (the released resource units are compared to the awaken agents' *requiredUnits* attribute).

Sometimes we need to know if a Binding is used to connect a service call or a signal. To do that we can use the function

$$\mathit{bindingType} : STEP \rightarrow \{call,signal\} \cup \{undef\} \tag{6.38}$$

that returns *call* if the Binding is linked to a Service, while it returns *signal* if the Binding is linked to a Wait step. If the step we are quering is not a ServiceControl step and therefore can't have a link to a Binding, then the function returns the value *undef*.

6.2.4 Rules for KLAPER ASM

In this section we will present the rules that define the semantics of the Steps that build the Behavior's graph of actions (that realize a Service). This means that we will analyze the way an execution flow of a service proceeds, describing its behavior in terms of *firing* of update rules that determine the way ASMs evolve in their state changes.

Steps correspond to the actions accomplished by ASM agents; therefore almost each Step has its own rule (only steps that are never instantiated, like Control and Reconfiguration, don't have their rule); these rules define how an ASM agent behaves when it arrives at each step; also we will define which is the impact of steps from the point of view of the two software quality attributes we are considering: performance and reliability.

Rule for the Start step

At the beginning of a Behavior we always have the same step: the Start step. The function of this step is simply to start the actions sequence of a Behavior, so its update rule is quite simple:

Rule **Start Step**
  if *active is start*(*name*, *in*, *out*)
    then *active* := *opposite*(*active*, *out*)

Effects on software quality attributes:

- performance: none.
- reliability: none, it can't fail.

Rule for the End step

On the opposite side of the Start step we have the End step that has the responsibility of closing the actions flow of a Behavior and therefore it stops the execution of an ASM agent. The update rule that describes this step is:

Rule **End Step**
  if *active is end*(*name*, *in*, *out*)
  then
    if (*called*(*parent*(*Self*)) = *true* **and** *isSynch*(*parent*(*Self*)) = *true*)
      or *isNested*(*Self*) = *true*
      then *mode*(*parent*(*Self*)) := *running*
        delete(*Self*)
    else *mode*(*Self*) := *undef*

where we are saying that if the current agent is a sub-agent (that is an agent born to execute a nestedBahavior or to handle a synchronous call) then the sub-agent has to return the control to its parent agent (the agent that created it) and then it can end its execution; if instead the current agent is a normal agent or if it is an agent born to handle an asynchronous call, then the agent is simply stopped (changing its state to *undef*).

Impact on software quality attributes:

- performance: none.
- reliability: none, it can't fail.

Rule for the Wait step

The Wait step has the role of waiting for the occurrence of a particular condition represented by an event; its update rule is:

Rule **Wait Step**
  if *active is wait*(*name*, *in*, *out*)
    then
      mode(*Self*) := *suspended*
      active := *opposite*(*active*, *out*)

When the agent's state is changed to *suspended*, instantly its execution is suspended; the agent will be restarted only when some other agent sends a signal to the agent itself changing its state to *running*; only at this point it is possible to change the currently active step to the "next" step and then going ahead with subsequent rules (more specifically a new update rule is triggered because we have linked update rules activation to the type of the current active step). The suspended agent when blocked in its execution stays on the current step (see the semantic of *mode* function in section 6.2.3) and when awakened its first action will be that subsequent to the change state action: in this case the first action executed will be setting a new active step. From the Wait semantics it is clear that if some signal is raised before the agent agent reaches the *suspended* mode during the rule execution, it will be lost; in practice if the ASM agent is still active and executing a step, changing its state to *active* has no effect because the agent is already in that condition. When a signal is raised before the agent has reached the *suspended* mode, as already said the signal is lost and the agent executing the Wait step rule will suspend its execution waiting for a new signal.

Impact on software quality attributes:

- performance: it doesn't consume processing power, but Wait step blocks an agent execution and therefore it adds time to the time needed to reach the final step.
- reliability: none, it can't fail.

Rule for the Branch step

The Branch step has the role to introduce the concept of "decision tree", that is it indicates two or more alternative ways that can be undertaken with a certain probability. The rule that represents this step is:

Rule **Branch Step**
  if $active\ is\ branch(name, in, out_i)$
   then
    if $probabilitySelected(out_1)$ **then** $active := opposite(active, out_1)$
    ...
    if $probabilitySelected(out_n)$ **then** $active := opposite(active, out_n)$

where $1 \leq i \leq n$ and obviously $\sum_{i=1}^{n} prob(out_i) = 1$ where with $prob(x)$ we express the probability of the Transition x.

Impact on software quality attributes:

- performance: none.
- reliability: none, it can't fail.

Rule for the Fork step

When we need to execute more parallel flows we must use the Fork step whose function is that of generating a variable number (this number is equal to the number of elements we have into the *out* list) of sub-agents that execute different actions simultaneously, temporarily suspending their parent agent until one or more sub-agents have completed their task (to see how and when the parent agent is reactivated see the update rule of the Join step). The update rule for the Fork step is:

Rule **Fork Step**
  if $active\ is\ fork(name, in, out)$
   then
    extend AGENT **with** $a_1 \ldots a_n$
      do forall $1 \leq i \leq n$
       $active(a_i) := opposite(active, out_i)$
       $parent(a_i) := Self$
       $called(a_i) := false$

$$mode(a_i) := running$$
$$mode(Self) := suspended$$

where n is equal to the cardinality of the *out* list. In the update rule the main agent creates n sub-agents; for each of them it initializes as the active step the first step of the related parallel flow and then sets the sub-agent mode to *running*. It is important to note that the parent agent is blocked at the Fork Step because it doesn't receive a new active state; it will be responsibility of sub-agents, when they will execute the related Join step, to set the parent agent to the new current step in consideration of what its sub-agents have done.

Impact on software quality attributes:

- performance: the parent agent is blocked but its sub-agents are running; therefore to the execution time required by the parent agent we have to add the time consumed by the slowest sub-agent from those required to unlock the related Join step.
- reliability: the parent agent can't directly fail on this step; but because sub-agents are running on other steps and consequently can fail, we can assume that also the parent agent fails if the number of failed sub-agents is equal to the number of sub-agents required to unlock the related Join step.

Rule for the Join Step

Sub-agents created into a Fork step when completed their tasks can either end or signal to their parent that they have finished their activities, reactivating the parent itself if needed. All that can be done with the Join step whose update rule is:

Rule **Join Step**
if $active\ is\ join(name, in, out, transitionsNeededToGo)$
then
$$subAgentsCompleted(parent(Self)) := subAgentsCompleted(parent(Self)) + 1$$
if $subAgentsCompleted(parent(Self)) = transitionsNeededToGo$
then
$$active(parent(Self)) := opposite(active, out)$$
$$mode(parent(Self)) := running$$

$delete(Self)$

where *Self* this time represents the agent executing the step, that means the sub-agent and not the parent agent. We have to note that if the parent agent have to be reactivated, this is done when the sub-agent is still running and only after the parent reactivation a sub-agent can terminate; this actions sequence should be done atomically, but even if this does not happen the update rule will be right because we will delay only the end of the sub-agent (that has completed all its activities and therefore can't cause any damage to the ASM).

Impact on software quality attributes:

- performance: none.
- reliability: the parent agent blocked on some Fork step can't fail directly, but we can assume that if a number of sub-agent fails, equal to the number of all possible sub-agents minus the number of sub-agents required to unlock the Join step, then also the father agent will fail. For example if we have a Fork step that can generate at most n sub-agents, then the related Join step fails if the number of failed sub-agents is greater than $n - transitionsNeededToGo$.

Rule for the Activity step

The step that can be considered the most important one between those composing KLAPER Behaviors is the Activity step that is responsible to model the execution of the real "actions" of the system. The update rule that describes the Activity step is quite complex because with this step we can realize not only simple activities, but also loops and nested behaviors:

Rule **Activity Step**
if *active is activity*(*name*, *in*, *out*, *repetitions*, *nestedBehavior*, *internalExecTime*, *internalFailProb*, *internalFailTime*)
then
if $nestedBahavior \neq undef$
then
extend AGENT **with** a
$active(a) := start(nestedBehavior)$
$parent(a) := Self$

$$
\begin{aligned}
&\qquad\quad called(a) := false\\
&\qquad\quad mode(a) := running\\
&\qquad\quad mode(Self) := suspended\\
&DoJob\\
&executedFromAgent(active, Self) := executedFromAgent(active, Self) + 1\\
&\textbf{if } executedFromAgent(active, Self) > repetitions\\
&\quad \textbf{then}\\
&\qquad active(Self) := opposite(active, out)
\end{aligned}
$$

where *repetitions* is a random variable. We have to note that loops are realized using the last *if* of the update rule that changes the active step (or better the active rule) only when the step itself has been repetitively executed for a random number of times equal to *repetitions*.

DoJob is a very particular function we defined but we did not present it in section 6.2.3 because its meaning is strictly related to Activity steps. It is represented as a nullary functions but actually it can depend on the parameters of the Service containing the Activity. It models the execution of all the activities of this step by an agent (activities that last a random time $internalExecTime$, fail with the probability $internalFailPorb$ and/or in the random time $internalFailTime$); definitively it is the place were we model that computation happens, time is spent and failures can occur.

Impact on software quality attributes:

- performance: the step consumes a time *internalExecTime* to execute all its duties; this is true also for loops (where *internalExecutionTime* is the time of a single run) and nested behaviors (where *internalExecutionTime* if expressed is the time of all the nested behavior, unless it is specified the *internalExecTime* of each step of the nested behavior, in this last case the attribute of the Activity is not valid). More precisely for hardware resources "consumes time" means Resource usage by a specific service; the way the resource is used depends on the specified scheduling policy. For Example for the most common scheduling policies we have:
  - fifo: each service exclusively holds the resource until it needs it and other services have to wait their turn. Enqueued services are served using

a First-In-First-Out policy. The *internalExecTime* is spent when the service holds the resource.

– round robin: all services are executed for a (small) quantum of time giving the impression of a parallel execution, but in reality each service is executed and exclusively holds the resource for a very short period of time. Usually all the services have the same priority and are served cyclically. The *internalExecTime* is spent for the little quantum of time, therefore the services looks like they are executing in parallel but in reality they are executing in a serial way and execution times are incremented (if compared to the fifo policy).

– infinite server: all services have all the resources needed and therefore they are all executed in parallel. The *internalExecTime* is immediately spent from the beggining of the service until its end even if there are more than one service executing over the resource.

– processor sharing: like infinite server policy all the services are executed in parallel but the execution capacity of the resource is equally distributed between all the services, therefore the real *internalExecTime* spent by each Activity is greater than the one specified.

- reliability: the step can fail with a probability expressed by *internalFailProb* (the probability that the Activity can fail, expressed with a stochastic distribution) or it can fail in a time expressed by *internalFailTime* (the time beyond wich the Activity can fail, expressed with a stochastic distribution). For loops and nested behaviors, *internalFailProb* or *internalFailTime* of the container Activity are considered only if they are not specified for contained Activities.

Just to give an example about how the execution of an Activity "consumes time", let's consider a Service running on a Resource that models a software component. The Service of a Resource modeling a software component certainly must model the usage of a cpu to accomplish some computations (like for example the operations needed to compute the result of a database query). To execute that Service or better an Activity of that Service we need[5] to request a service to a Resource representing

[5]This is only one of the possible patterns that can be used to model such a situation, other patterns can be developed by software models specialists

a physical cpu (therefore representing an hardware component); this service request is done using a ServiceControl modeling a synchronous call to the basic Service offered by the cpu Resource. The Service of the cpu resource is modeled using a single Activity that has a specific *internalExecTime* to model the service time of the resource (often this service time is function of some formal parameters of the Service that can be used to specify for example the number of elementary operations needed by the requester of the service). Because the service call (and therefore the ServiceControl) is synchronous, the Service running on the Resource modeling the software component has to wait until the requested service is completed; this means that the Resource modeling the hardware component could be busy (according to the specified scheduling policy, see the discussion of the impact of the Activity step on performance) and could not satisfy other requests. This can be translated into an utilization of some kind of servant (or processor) in quality analysis models. In figure 6.6 we can see an example of the just described scenario.

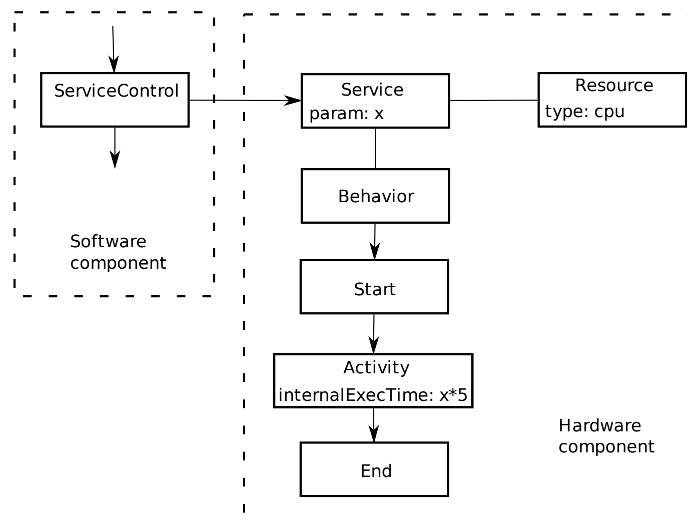

Figure 6.6. Example of “time consumption” for a KLAPER ServiceControl (that is a special case of an Activity)

Rules for Acquire and Release steps

Acquire and Release steps have update rules quite simple:

Rule **Acquire Step**
if *active is acquire*(*name*, *in*, *out*, *resourceUnits*, *resource*)
then
DoJob
if *availableResources*(*resource*) - *resourceUnits* ≤ 0
then
requiredResource(*Self*) := *resource*
requiredUnits(*Self*) := *resourceUnits*
mode(*Self*) := *suspended*
availableResources(*resource*) := *availableResources*(*resource*) - *resourceUnits*
opposite(*active*, *out*)

where *DoJob* is the same as in Activity. The Acquire step update rule checks if there are enough resource units for the required resource; if this is not the case it sets the agent's *requiredResource* and *requiredUnits* attributes and put itself in suspended mode waiting for the required resources; when the required resources are available, the agent is awaken and the resources availability is decremented. Note that if the required resource was already available the agent never put itself to the *suspended* mode but immediately acquire the resources.

Rule **Release Step**
if *active is release*(*name*, *in*, *out*, *resourceUnits*, *resource*)
then
DoJob
availableResources(*resource*) := *availableResources*(*resource*) + *resourceUnits*
mode(*findAgentRequiring*(*resource*, *resourceUnits*)) := *running*
opposite(*active*, *out*)

where *DoJob* is the same as in Activity. This rule simply increments the number of available resources and wakes up an agent waiting for a number of resources less or equal to *resourceUnits*; if there isn't any agent waiting for the resource the *mode*() function does nothing.

Impact on software quality attributes:

- performance: the same as Activity even if *internalExecTime* is not shown into the rule, but actually Acquire and Release are specializations of Activity.

- reliability: the same as Activity even if *internalFailProb* and *internalFailTime* are not shown into the rule, but actually Acquire and Release are specializations of Activity.

Rules for CreateBinding and DeleteBinding steps

Here we have to made the same considerations already done for the previous steps, therefore the update rules are quite simple and are:

Rule **CreateBinding Step**
  if *active is createBinding*(*name*, *in*, *out*, *sourceStep*, *targetService*)
    then
      DoJob
      opposite(*active*, *out*)

Rule **DeleteBinding Step**
  if *active is deleteBinding*(*name*, *in*, *out*, *sourceStep*, *targetService*)
    then
      DoJob
      opposite(*active*, *out*)

where *DoJob* one more time represents the activities done by the agent into this step. In particular, in the *CreateBinding Step* update rule, *DoJob* creates a new Binding that has as source the *sourceStep* Step and as target the *targetService* Service. Similarly in the *DeleteBinding Step* update rule *DoJob* deletes a Binding element between the *sourceStep* Step and the *targetService* Service.

Impact on software quality attributes:

- performance: see Acquire and Release.
- reliability: see Acquire and Release.

Rule for the ServiceControl step

The ServiceControl step is a specialization of the Activity Step used to request services, therefore it realizes interactions between resources. Since ServiceControl is a specialization of Activity in this step we can find all the properties of the parent meta class plus the functionalities needed to call services. The update rule is:

Rule **ServiceControl Step**

if $active$ *is* $serviceControl(name, in, out, resourceType, serviceName,$
$isSynch, bindign, actualParam)$

then

do Activity Step job

if $bindingType(active) = signal$

then

$mode(target) := running$

else

extend AGENT **with** a

$active(a) := start(calledBehavior(active))$

$parent(a) := Self$

$called(a) := true$

$mode(a) := running$

if $isSynch = true$

then

$mode(Self) := suspended$

$opposite(active, out)$

where with *target* we refer to the Wait step linked to this signal. In practice when an agent enters this step it executes first all the operations typical of an Activity, then if we are in the signal configuration a sleeping agent is awaken, whereas if we are in the service call configuration a new agent is created that handles the Behavior of the called service.

Impact on software quality attributes:

- performance: if it represents a signal it doesn't consume time, but if instead it represents a service call it consumes time only if it is a synchronous service call because it has to wait the called service response (while in case of an asynchronous service call this is not done).

- reliability: if this step models a signal[2], the failure of the signaled step has consequence only on the reliability of that step (see failure conditions for the

[2]We can understand if a ServiceControl models a service call or a signal from the linked Binding. If the Binding has a value for the *call* attribute that means it is modeling a service call, otherwise if it has a value for the *signal* attribute that means it is modeling a signal.

Wait step); that means that the failure of the signaled step doesn't cause a failure of this step, but however this step can always fail on its own. If considered as a service call a failure of the called service causes a failure in this step for a synchronous call; for an asynchronous call a failure of the called service causes a failure of the caller only if the *dependsOn* attribute is set to *true*, if it is set to *false* a called service failure has no consequences on the caller.

Chapter 7

Transforming to DTMC

As we already saw in section 3.2, DTMC can be used as a software quality model aimed at evaluating reliability. In this chapter we will see the rules used to map a KLAPER model to a DTMC model. Before doing so we will analyze the DTMC meta model that is the target of our transformation.

7.1 The DTMC meta model

The DTMC meta model showed in figure 7.1 has been built starting from the general concepts characterizing DTMCs, like states, transitions and so on. However, these are not the only concepts we can find into the DTMC meta model; indeed we have added also some foreign concepts. Some of these foreign concepts, like the ExternalReference meta class, have been introduced to simplify the subsequent model-to-text transformation, while some other concepts, like the completionModel and the internalFailProb attributes of the State meta class, have been introduced to express some concepts typical of the KLAPER meta model. With this approach we have a meta model which we may refer as a hybrid meta model, because it has a lot of concepts of a canonical DTMC plus some other concepts coming from the KLAPER world. The task of purging the DTMC hybrid meta model from all the foreign concepts, leaving only the DTMC fundamental ones, is left to the model-to-text transformation with its mapping rules. We followed this kind of approach because it strongly simplifies the model-to-text transformation and the related code generation.

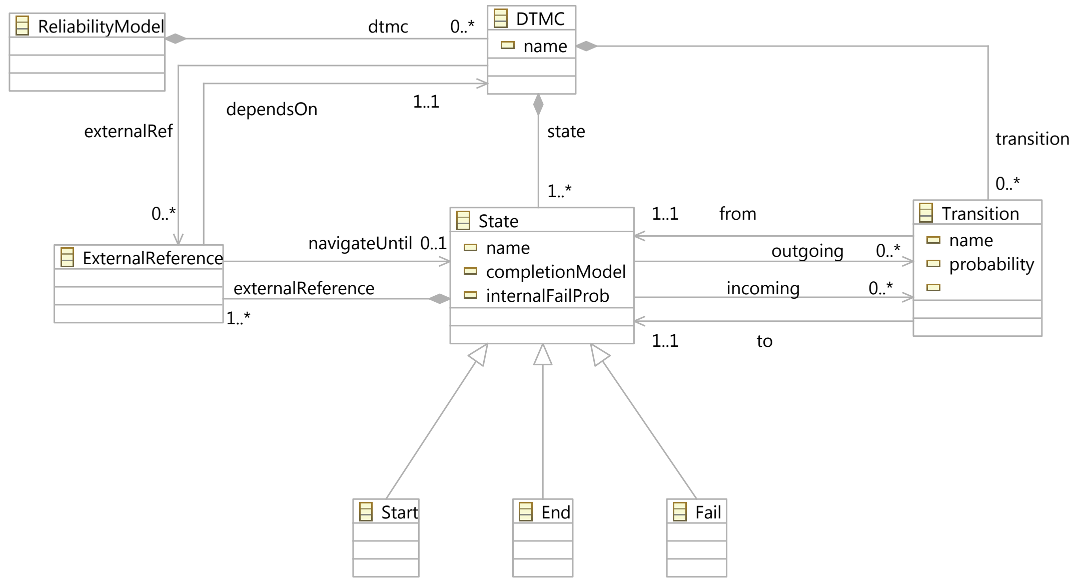

Figure 7.1. DTMC hybrid meta model

The DTMC meta model is quite simple if compared to the KLAPER meta model. In this section we will see an overview of the meta model; for a more detailed description of all classes' attributes see [8].

The DTMC meta model consists of the following meta classes:

- **ReliabilityModel**: this meta class is only a container added to simplify the meta model handling, but it hasn't any conceptual value for DTMC.

- **DTMC**: meta class that represents a DTMC into the model. Models can be composed by many DTMCs and all the DTMCs of a model can be classified into two different types: fundamental DTMCs, that represent the services into our system, and support DTMCs, that represent particularly sensitive portions of the system. Maintaining this classification is very effective to analize critical sections of a system that can be isolated into a support DTMC.

- **Transition**: meta class that represents the transition from one state to the other of a DTMC. It is characterized by the probability that the transition happens and the source and target states at the extremity of the transition.

- **State**: this meta class represents a state into the DTMC. We can identify two kind of states: those that model the execution of a simple action (called simple states) and those (called complex states) that model the request for services. States are linked between them using Transitions. Some States can fail so the meta class has an attribute to define the failure probability and an attribute to describe how a State fails if it is a complex state (that is the completion model used when we have parallel calls). Simple states failures depend only on their internalFailProb attribute failure probability, while complex states failures depend on the failure probabilities of the called entities.

- **Start**: it is a particular State that can never fail and that is used as starting point of a DTMC.

- **End**: it is an absorbing state that represents the correct end of a DTMC; that is, it is the state where we go when all into the system runs properly and so we can reach a correct ending point.

- **Fail**: it is an absorbing state that represents a failure condition of the system (we have a single Fail state for all the failure conditions of the system); it is the state where we fall when something wrong happens into the system and a correct end is impossible to reach.

- **ExternalReference**: this meta class is used by complex states to represent a reference to a state that the referencing state depends on. The dependency can be synchronous or asynchronous (see the attribute *dependsOn*) and can be conditioned on reaching a particular State if we are modeling a system with signals.

7.2 KLAPER to DTMC transformation, concepts

Before going deep into the code of transformation rules that implement the mapping from the KLAPER meta model to the hybrid DTMC meta model, we will present now some general concepts applied during the transformation. They show the key concepts underlying the transformation itself:

- KLAPER Workloads are not considered in this work. Even if someone can argue their influence on reliability, in this work we assume that this influence can be considered irrelevant.
- KLAPER Resources haven't any correspondence into the hybrid DTMC meta model; this is because into the hybrid DTMC all is related to services and there isn't the concept of a group of services.
- Each KLAPER Service is mapped to a DTMC DTMC meta class (KLAPER Resources has not any equivalent concept into DTMC).
- Each DTMC has exactly one Fail state that represents the absorbing condition of a failure of the related DTMC.
- Each KLAPER Step is mapped to a DTMC States.
- Each KLAPER Start step is mapped to a DTMC Start state.
- Each KLAPER End step is mapped to a DTMC End state.
- Each KLAPER Activity is mapped to a DTMC State. If the Activity is repeated only once and doesn't have any nested behavior, the failure probability of the step is used to define the probability of the transition to the Fail state of the DTMC which the new state is part of. Otherwise a new DTMC, intended to model the nested behavior, is created with the related external reference and the transition probability to the Fail state is set to 0.
- Each KLAPER Wait step is mapped into a simple DTMC state that later will be linked to the related signal source through an ExternalReference meta class instance (placed into the DTMC where is located the signaling state).
- Each KLAPER ServiceControl step is mapped just like Activity steps, but this time the external reference (to the DTMC that models the called service) depends also from the synchronization of the call: signals and asynchronous calls with the *dependsOn* attribute set to *false* can't have an external reference because their asynchronous nature doesn't imply any dependency from any other service; that means that a failure of the signaled element or of the called service does not cause a failure into the calling entity; on the contrary

synchronous calls and asynchronous calls with the *dependsOn* attribute set to *true* always have an external reference (even if they are repeated only once and don't have a nested behavior) because a failure of the called service can cause a failure in the caller entity.

- Each KLAPER Fork step is mapped to DTMC built as follows: a simple state is created corresponding to the root of the fork. Then for each path of the fork we create a fake KALPER Service that is transformed using the rule previously presented for general Services. The simple state created as fork root has an external reference to each DTMC coming from fake services.
- Each KLAPER Branch step is mapped to a simple DTMC state (branches are managed using DTMC states transitions with their probabilities).
- Each nested behavior is mapped to a new DTMC DTMC meta class.
- DTMC created from Activities with repetitions have a structure that depends on the probability distribution function used to represent repetitions. In the actual transformation all the probability distribution functions are reduced to only two cases: geometric distribution function and histogram distribution function. All constant distribution are reduced to an histogram distribution with a single sample with probability 1.0 and as value the value of the constant original distribution; all other distribution are reduced to a geometric distribution having the same mean of the original distribution.
- KLAPER Join steps don't have any direct mapping because every DTMC state can be a join point if it collects many paths coming from the same fork state.

7.3 KLAPER to DTMC transformation, implementation

In the previous section we outlined some general concepts followed designing the transformation from KLAPER to the hybrid DTMC meta model. Now we want better specify the transformation rules mentioned before; to do this we will see a

real implementation of the general rules just seen done using the Xtend scripting language we already described in section 4.3.3; it is an OCL like language that is not a complete implementation of the QVT standard, but it is very close to it. See [45] to have more details about the Xtend tool.

In this chapter we will see only the most relevant portions of the transformation. To see the complete transformation code see appendix C.

The starting point of the transformation is the function klaper2dtmc():

```
23 /*
24         Starting point for the transformation (this extension is invoked directly
               from the workflow).
25         PLEASE NOTE: The input KlaperModel is supposed to be well formed for the
               purpose of reliability analisys with DTMC!
26 */
27 dtmc::core::ReliabilityModel klaper2dtmc(klaper::core::KlaperModel m):
28         m.transformModel();
```

that simply calls the transformModel() function over a KlaperModel instance; it is only an access point to the real transformation.

Then the transformModel() function starts the true transformation of the KLAPER model into a DTMC model:

```
30 /*
31         Creates a ReliabilityModel from the input KlaperModel
32 */
33 private create dtmc::core::ReliabilityModel newModel transformModel(klaper::core::
       KlaperModel m):
34         newModel.dtmc.addAll(m.resource.offeredService.transformService(newModel))
               ->
35         m.resource.offeredService.behavior.step.typeSelect(klaper::core::
               ServiceControl).select(e|e.binding.signal!=null).collect(e|e.linkWait(
               newModel))->
36         newModel;
```

where on line 34 we transform each KLAPER service into a DTMC DTMC meta class instance and then on line 35 we link ExternalReference instances with State instances (we will see the linkWait() function later) starting from KLAPER signals.

The linkWait() function is responsible to create ExternalReference instances with related attributes to set reliability dependency for synchronous and asynchronous KLAPER ServiceCalls and for Wait steps

```
39 private Void linkWait(klaper::core::ServiceControl s, dtmc::core::ReliabilityModel
       m):
40                 let extRef=new ExternalReference:
41                 let waitSt=s.binding.signal.retrieveWaitState(m):
42                 let scSt=s.retrieveServiceControlState(m):
```

```
43                 extRef.setDependsOn((dtmc::core::DTMC)scSt.eContainer)->
44                 extRef.setNavigateUntil(scSt)->
45                 waitSt.externalReference.add(extRef);
```

where the setDependsOn() function sets the *dependsOn* attribute that defines if a state, that requires the services offered by another state (using an ExternalReference), fails in case of a called state failure. The setNavigateUntil() completes the ExternalReference initialization setting the state that the current state refers to (the referred state must be derived from a KLAPER Wait); if the current state can't be reached because of a failure also the referred state can be considered as failed because speaking in KLAPER terms it will never be awaken.

But before setting ExternalReferences we have to transform each KLAPER Service into a DTMC instance using the function transformService()

```
46  /*
47          Creates a DTMC from each Service offered by a given Klaper Resource (
                Workloads are not useful for reliability analsisys).
48          A Fail state is added to the Markov Chain at once.
49  */
50  private create dtmc::core::DTMC newDtmc transformService(klaper::core::Service s,
        dtmc::core::ReliabilityModel m):
51          let f = new Fail:
52          f.setName("Fail")->
53          newDtmc.setName( ((klaper::core::Resource)s.eContainer).name+"_"+s.name )->
54          newDtmc.state.add(f)->
55          newDtmc.state.addAll(s.behavior.step.collect(e|transformStep(e,newDtmc,m)))
                ->
56          newDtmc.state.select(e|e.externalReference.size==0).collect(e|
                updateTransitionProbabilities(e))->
57          newDtmc.state.removeAll(newDtmc.state.select(e|e.name==null))->
58          newDtmc;
```

where each DTMC instance is created with an absorbing fail state named "Fail" and a list of states directly converted from KLAPER steps; updateTransitonProbabilities() is a function that computes all the transition probabilities of a state considering the failure probability of the state itself (that is the probability to activate the transition to the "Fail" absorbing state), but it is applied only to those states that don't have ExternalReferences (e.externalReference.size==0), that means to KLAPER steps that don't do service calls or signals.

As already said in the previous section, KLAPER Workload meta class is not considered into the DTMC transformation because workloads are assumed to be not relevant for reliability evaluation[1]. Note that we also don't care about the

[1]This is an assumption done in this thesis and even if results obtained in this way are enough

KLAPER concept of Resource; in DTMC all is related to services and therefore some KLAPER concepts have no mapping into the transformation rules. Also the standard KLAPER Behavior has no relevance for reliability, except for nestedBehaviors, but we will see them later.

Once the DTMCs of the system have been created we only need to transform each KLAPER Step into its corresponding DTMC State representation.

Before describing the transformations applied to KLAPER Steps we have to introduce some concepts about how the KLAPER Join Step is mapped in the transformation from KLAPER to DTMC. KLAPER Join has not a real mapping into any meta class of the DTMC meta model; simply it is mapped into any DTMC State due to the presence of the *completionModel* attribute; that means that any DTMC State can represent a KLAPER Join Step. Because every State can map a Join (see figure 7.2 for an example), in every DTMC State transformation rule we will find the stepBelongsToForkJoin() function that has the aim to establish if a particular State is contained between a Fork-Join pair (with the possibility of nesting). This is very useful when converting a Fork; indeed, as we will see later, the DTMC meta model represents the sequence of actions between a KLAPER Fork-Join pair with a support DTMC (see section 7.1 about support DTMCs) and the stepBelongsToForkJoin() function is very useful to establish where a support DTMC ends.

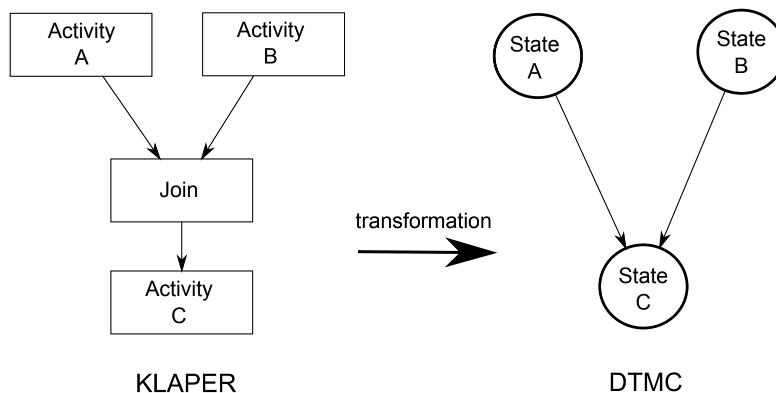

Figure 7.2. Transformation of a KLAPER Join step into the DTMC meta model. DTMC doesn't have a specific entity to represent join condition.

Transformations for KLAPER Start and End steps are so simple that they don't

accurate, the impact of workload on reliability can give even more accurate results

require any explanation.

```
72  /*
73          Creates a DTMC Start state from a Klaper Start step
74  */
75  private create dtmc::core::Start newState transformStep(klaper::core::Start s, dtmc
        ::core::DTMC d,  dtmc::core::ReliabilityModel m):
76          newState.setName(s.name)->
77          d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.
              select(x|x.from==s).collect(e|e.transformTransition(d,m)))->
78          newState;
79
80  /*
81          Creates a DTMC End state from a Klaper End step
82  */
83  private create dtmc::core::End newState transformStep(klaper::core::End s, dtmc::
        core::DTMC d,  dtmc::core::ReliabilityModel m):
84          newState.setName(s.name)->
85          d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.
              select(x|x.to==s && x.from.metaType!=Join).collect(e|e.
              transformTransition(d,m)))->
86          newState;
```

Also the transformation for the KLAPER Wait step is quite simple

```
124         Creates a DTMC State from a KLAPER Wait step
125 */
126 private create dtmc::core::State newState transformStep(klaper::core::Wait s, dtmc
        ::core::DTMC d,  dtmc::core::ReliabilityModel m):
127         (s.stepBelongsToForkJoin(0,{})==false)?
128         (
129                 newState.setName(s.name)->
130                 newState.setCompletionModel("OR")->
131                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                      transition.select(x|(x.to==s&& x.from.metaType!=Join)||x.from==
                      s).collect(e|e.transformTransition(d,m)))->
132                 newState
133         ):
134         {};
```

where the only thing to note is the state completion model set to “OR” to indicate that the execution flow can go over the state without waiting any other incoming Transition.

To close the list of simple transformations we can see the KLAPER Branch step transformation that simply creates a new named DTMC State

```
225 /*
226         Creates a DTMC State corresponding to a Klaper Branch Step.
227 */
228 private create dtmc::core::State newState transformStep(klaper::core::Branch s,
        dtmc::core::DTMC d,  dtmc::core::ReliabilityModel m):
```

```
229
230         (s.stepBelongsToForkJoin(0,{})==false)?
231         (
232                 newState.setName(s.name)->
233                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                    transition.select(x|(x.to==s&& x.from.metaType!=Join)||x.from==
                    s).collect(e|e.transformTransition(d,m)))->
234                 newState
235         ):
236         {};
```

the only thing to note here is that Branch can have multiple outgoing Transitions, therefore line 233 will add many DTMC Transitions as those that go out from the KLAPER Branch step.

The transformation from the KLAPER Fork step is the first with a bit of complexity so far

```
203 /*
204         A top-level Fork (i.e. a Fork which is not nested in another Fork-Join
                pattern) is mapped into a new State which points (by means of n
                ExternalReferences)
205         to the DTMCs corresponding to the n paths starting from such Fork. Each
                path is first transformed into a Klaper Service (by means of
                transformForkPathToService())
206         and then such Service is transformed into a DTMC (createExternalReference()
                 does this job).
207
208 */
209 private create dtmc::core::State newState transformStep(klaper::core::Fork s, dtmc
        ::core::DTMC d,  dtmc::core::ReliabilityModel m):
210         (s.stepBelongsToForkJoin(0,{})==false)?
211         (
212                 let pathServices = (List[klaper::core::Service]){}://Each path
                    starting from Fork is mapped to a Service
213                 s.out.collect(e|pathServices.add(e.to.transformForkPathToService(s,
                    m)))->
214
215                 newState.externalReference.addAll(pathServices.collect(e|e.
                    createExternalReference(m)))->
216                 newState.setName(s.name)->
217                 newState.setCompletionModel(s.retrieveJoinFromFork(0,{}).in.size==s
                    .retrieveJoinFromFork(0,{}).transitionsNeededToGo?"AND":"OR")->
218                 d.state.add(newState)->
219                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                    transition.select(e|e.to==s).collect(e|e.transformTransition(d,
                    m)))->
220                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                    transition.select(e|e.from==s.retrieveJoinFromFork(0,{})).
                    collect(e|e.transformJoinTransition(d,newState,m)))
221         ):
```

```
222         {};
```

the idea here is to create a State with an ExternalReference to a DTMC derived from fake Services artificially built for each path (from each path we build a new Service with a new Behavior composed by steps taken from the path itself) that starts from the Fork step (see lines 212 and 213); see figure 7.3 to better understand how the transformation works. After that the transformation sets the completion model of the State; this is done comparing the number of outgoing paths starting from the Fork step with the number of required completion set on the Join related to the Fork under exam (the completion model is set to “AND” if all the fork paths are required to complete to have the permission to exit from the Join step, otherwise it is set to “OR”). To complete the work we only need to transform Transitions (see line 219 and 220).

Transforming a KLAPER Activity is a little bit more complex than the steps seen until now:

```
89  /*
90          Creates a DTMC State from a Klaper Activity. A transition towards the Fail
                state (whose probability is elicited from the Activity's
                internalFailProb attribute)
91          is added to the Chain only in the case the Activity is repeated only once
                and does not contain a nested Behavior. In all the other cases, the
                Activity is mapped into
92          a new DTMC (by means of transformActivity()) and the State created here
                contains an ExternalReference pointing to such DTMC.
93  */
94   private create dtmc::core::State newState transformStep(klaper::core::Activity s,
         dtmc::core::DTMC d,  dtmc::core::ReliabilityModel m):
95
96          (s.stepBelongsToForkJoin(0,{})==false)?
97                  (
98                          s.transformActivityRepetitions()->
99                          s.isASimpleActivity()?
100                         (
101                                 newState.setName("ACT_"+s.name)->
102                                 d.transition.addAll(((klaper::core::Behavior)s.
                                        eContainer).transition.select(x|(x.to==s&& x.
                                        from.metaType!=Join)||x.from==s).collect(e|e.
                                        transformTransition(d,m)))->
103                                 d.transition.add(s.transitionToFail(d,m))->
104                                 newState
105                         )
106                         :
107                         (
108                                 m.dtmc.add(s.transformActivity(m))->
109                                 (
```

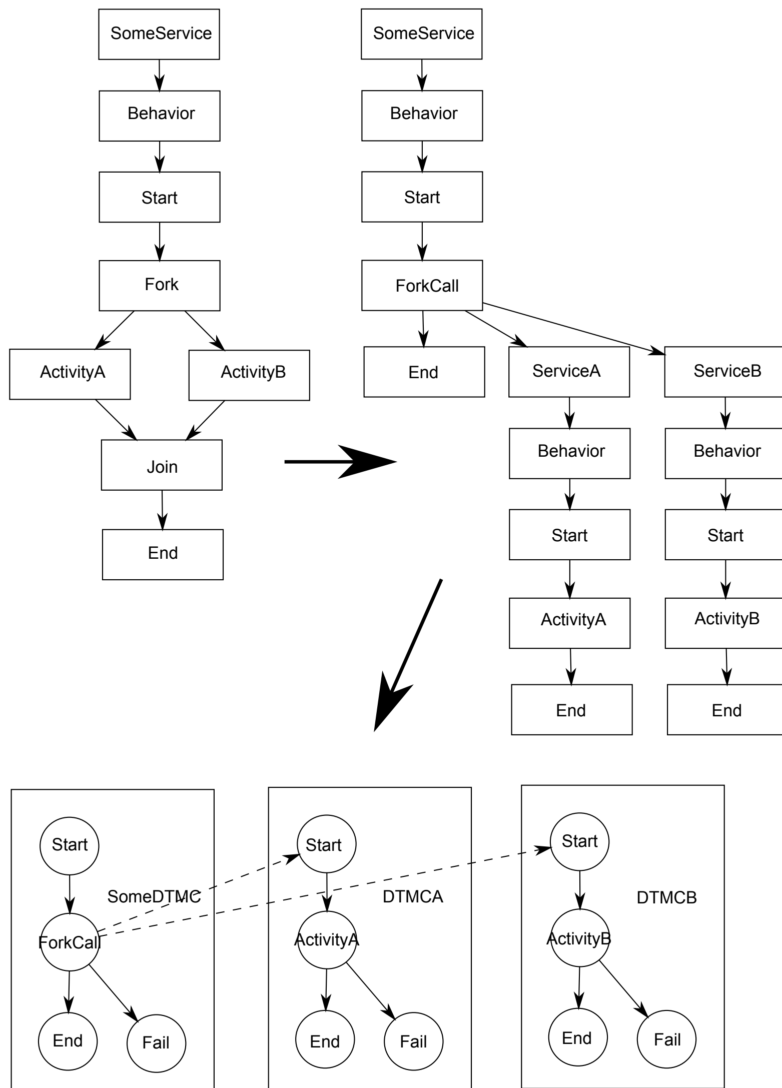

Figure 7.3. Transformation of a KLAPER Fork step into the DTMC meta model. Each path of the fork is mapped into an external DTMC.

```
110                                    let extRef=new ExternalReference:
111                                    extRef.setDependsOn(s.transformActivity(m))
                                           ->
112                                    newState.setInternalFailProb(0.0)->
113                                    newState.externalReference.add(extRef)
114                             )
115                             ->
116                             newState.setCompletionModel("OR")->
```

```
117                                     newState.setName(s.name)->
118                                     d.transition.addAll(((klaper::core::Behavior)s.
                                             eContainer).transition.select(x|(x.to==s&& x.
                                             from.metaType!=Join)||x.from==s).collect(e|e.
                                             transformTransition(d,m)))->
119                                     newState
120                          )
121               )
122               :{};
```

if the Activity is an activity without nested behaviors and without repetitions, then it is directly mapped to a DTMC State with a Transition to the Fail state of the same DTMC; where the probability to go to the Fail State is computed starting from the KLAPER Activity *internalFailProb*, using the function transitionToFail() (see lines from 101 to 104). If instead the KLAPER Activity has a nested behavior or has more than one repetition then the transformation rule creates a new DTMC State with an ExternalReference to a new DTMC (lines from 108 to 114) that will map the KLAPER Activity; in figure 7.4 we have an example of the possible transformations for a simple activity (a), an activity with repetitions (b) and an activity with a nested behavior (c). The new DTMC State is created using the function s.transformActivity(m)); it is not reported here due to its big dimension and considering the fact that it is not really relevant to understand the transformation between meta models because it simply map the Activity to a States graph according to the probability distribution function chosen for the KALPER Activity *repetitions* attribute.

As we already saw in the previous chapter, in KLAPER Activity has a very important specialization called ServiceControl that is used to request some services to other resources or to raise signals. Here we see how ServiceControl is transformed from KLAPER to DTMC:

```
137 /*
138         Creates a DTMC State from a Klaper ServiceControl. In the case the
                ServiceControl is repeated just once, newState contains already an
                ExternalReference pointing to
139         the DTMC representing the invoked service and its internalFailProb is set
                with the same value as the original ServiceControl. In all the other
                cases,
140         the ServiceControl is mapped into a new DTMC (by means of
                transformServiceControl()) and the State created here contains an
                ExternalReference pointing to such DTMC.
141 */
142 private create dtmc::core::State newState transformStep(klaper::core::
        ServiceControl s, dtmc::core::DTMC d, dtmc::core::ReliabilityModel m):
```

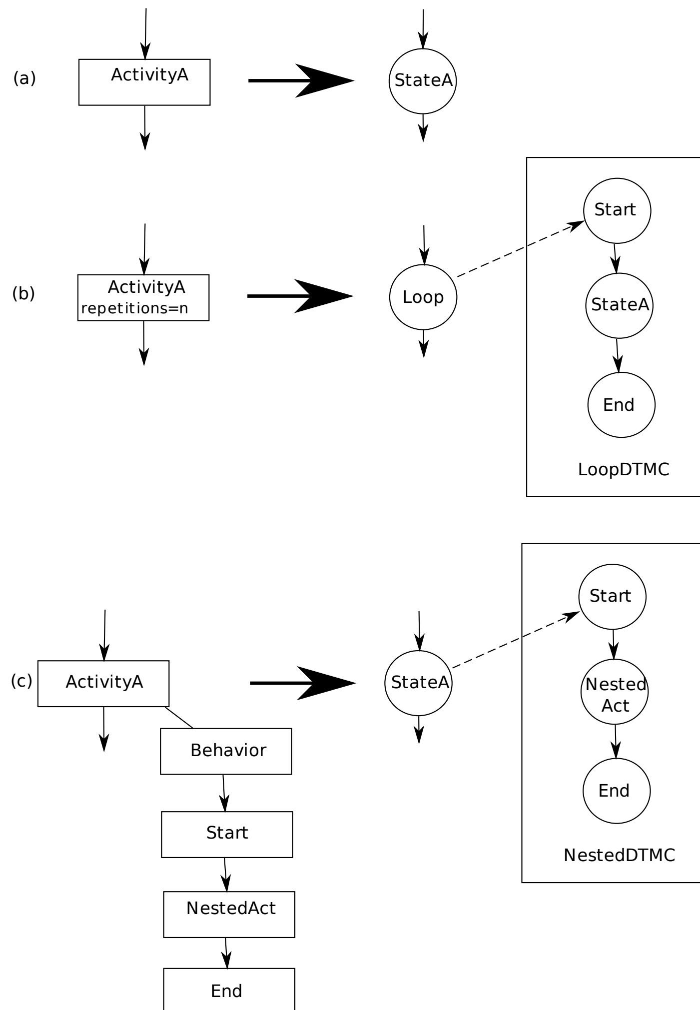

Figure 7.4. Transformation of a KLAPER Activity step into the DTMC meta model. (a) simple Activity (b) Activity with repetitions (c) Activity with a nested behavior.

```
143            (s.stepBelongsToForkJoin(0,{})==false)?
144                    (//belongsToForkJoin=false
145                            s.transformActivityRepetitions()->
146
147                            s.isASignalServiceControl()?
148                            (//signal
149
150                                    newState.setName( s.name )->
151                                    newState.setInternalFailProb(s.internalFailProb.
                                           getMean())->
```

```
152                             d.transition.addAll(((klaper::core::Behavior)s.
                                    eContainer).transition.select(x|(x.to==s&& x.
                                    from.metaType!=Join)||x.from==s).collect(e|e.
                                    transformTransition(d,m)))->
153                             d.transition.add(s.transitionToFail(d,m))->
154                             newState
155                     )//signal
156                     :
157                     (//call
158 
159                      (s.isSynch==false&&s.dependsOn==false)?
160                      (//synch=false
161                             newState.setName(s.name)->
162                             d.transition.addAll(((klaper::core::Behavior)s.
                                    eContainer).transition.select(x|(x.to==s&& x.
                                    from.metaType!=Join)||x.from==s).collect(e|e.
                                    transformTransition(d,m)))->
163                             d.transition.add(s.transitionToFail(d,m))->
164                             newState
165                      )//synch=false
166                      :
167                      (//synch==true
168                             s.isASimpleServiceControl()?
169                             (//simpleSC
170                                     newState.setName("SC_"+s.name)->
171                                     (
172                                             let extRef=new ExternalReference:
173                                             extRef.setDependsOn(s.binding.call.
                                                    transformService(m))->
174                                             newState.setInternalFailProb(s.
                                                    internalFailProb.getMean())->
175                                             newState.externalReference.add(
                                                    extRef)->
176                                             newState.setCompletionModel("OR")
177                                     )->
178                                     d.transition.addAll(((klaper::core::
                                            Behavior)s.eContainer).transition.
                                            select(x|(x.to==s&& x.from.metaType!=
                                            Join)||x.from==s).collect(e|e.
                                            transformTransition(d,m)))->
179                                     newState
180                             )//simpleSC
181                             :
182                             (//complexSC
183                                     m.dtmc.add(s.transformServiceControl(m))->
184 
185                                     (
186                                             let extRef=new ExternalReference:
187                                             extRef.setDependsOn(s.
                                                    transformServiceControl(m))->
188                                             newState.setInternalFailProb(0.0)->
```

```
189                                                             newState.externalReference.add(
                                                                    extRef)
190                                                     )
191                                                     ->
192                                                     newState.setCompletionModel("OR")->
193                                                     newState.setName(s.name)->
194                                                     d.transition.addAll(((klaper::core::
                                                            Behavior)s.eContainer).transition.
                                                            select(x|(x.to==s&& x.from.metaType!=
                                                            Join)||x.from==s).collect(e|e.
                                                            transformTransition(d,m)))->
195                                                     newState
196                                         )//complexSC
197                                 )//synch=true
198                                 )//call
199                     )//belongsToForkJoin=false
200                     :
201                     {};
```

we have three different behaviors according to the meaning of the ServiceControl:

- ServiceControl as a signal (lines from 147 to 155): a new DTMC State is created with related Transitions including the one to the DTMC's Fail State. The new State has an *internalFailProb* attribute computed from that of the related KLAPER ServiceControl; this attribute is not the real failure probability of the State (which is instead correctly expressed using the Transition to the Fail State of the same DTMC), but it is used to simplify the computation of the failure probability of a Wait State potentially linked to the newly created State.

- ServiceControl as an asynchronous call (lines from 159 to 165): a new DTMC State is created with related Transitions including the one to the DTMC's Fail State.

- ServiceControl as a synchronous call (lines from 167 to 196): here we have to specify two different sub cases: simple ServiceControl (for ServiceControls that don't have repetitions or nested behaviors but simply make a synchronous call to another service) and complex ServiceControl (for ServiceControls that do not simply make a synchronous call to another service but also have the *repetitions* attribute greater than 1 and/or have a nested behavior).

  – Simple ServiceControl (see lines from 169 to 180): a new DTMC State with an ExternalReference is created. The new State can't fail, indeed it

hasn't any Transition to the Fail State of its DTMC (there is no transitionToFail() function called). The *internalFailProb* attribute (set at line 174) will be used later in the M2T transformation to set the failure probability of the referenced DTMC. The ExternalReference point to another DTMC created from the KLAPER Service called (see the "binding" element of line 173) by the KLAPER ServiceCall (we have already seen how a KLAPER service is transformed into a DTMC).

– Complex ServiceControl (see lines from 182 to 196): this case is identical to the simple ServiceControl case except for two things: the first (see line 188) is that the *internalFailProb* attribute is set to 0 (this is because all the responsibilities of a failure are demanded to the newly created DTMC), the second (see line 183) is that this time a new DTMC is created to handle all the activities accomplished by the complex ServiceControl and the true call will be done by one of the States of this new DTMC. As already done with the transformActivity() function we do not present here the transformServiceControl() function because it simply apply a State configuration that is required to build a correct DTMC, but has really relevant for the description of the meta model transformation rules.

We have not presented all the transformation rules used to go from the KLAPER meta model to the DTMC one; we only presented those excerpts that are more relevant to the aim of understanding how the transformation works. To see the full transformation and to have more details about it take a look at [8].

7.4 Using the DTMC meta model

This section should be named "Using the DTMC model" because after applying the KLAPER to DTMC transformation, starting from a KLAPER model we have a real DTMC model; in the title we spoke about meta models because transformations rules are defined between meta models and models are only an artifact used as input and obtained as result.

A DTMC model is useless until it is converted to an input file for some tool. In our case we want to transform (and the word "transform" has not been chosen

accidentally) our model into an input file for the SHARPE tool (see [53]). The transformation to the SHARPE input file is a typical example of a M2T transformation where from a model we obtain a text file; due to the space required by the model to text transformation we can't explicitly analyze it here, but we prefer to concentrate our attention to the intermediate meta model (KLAPER) and to the model to model transformation that start from it; however you can find the complete model to text transformation, realized using Xpand, in [8] where you will find also a complete explanation about the use of the SHARPE tool and about interpreting the obtained results. In chapter 10 we will see a complete application of the transformation chain related to the DTMC meta model as explained until now; starting from a KLAPER model we will transform it into a DTMC model (M2M transformation), from this model we will generate an input file (M2T transformation) that submitted to the SHARPE tool will return some results to analyze (software quality analysis) and may be to use to give some feedback about the starting model.

Chapter 8

Transforming to SimJava

8.1 The SimJava meta model

With SimJava, like with DTMCs, we can evaluate reliability; moreover, like with LQN, with SimJava we can also evaluate performance. But while DTMC and LQN use analytical solvers to compute the required indexes, SimJava adopts a different kind of approach based on simulation. That means that from KLAPER we will not build some analysis model expressed in a particular formalism (required by some kind of tool as input), rather we generate real executable Java code, based on the SimJava library, that implements the behavioral semantic of KLAPER .

In any case in a model driven approach to build a simulator we need to define a meta model of simulation models that we want to generate; you can see such a SimJava based meta model in figure 8.1.

Here we will present an overview of meta classes of the meta model based on SimJava (if you want to have more details about this meta model see [9]).

The SimJava based meta model consists of the following meta classes:

- **SimModel**: this meta class is the usual container like those already seen for other meta models.
- **Workload**: it represents the workload of the system, that is it models the way actors interact with the system; in SimJava this means a set of Steps (see later) that are executed with a specified flow.
- **OpenWorkload**: this meta class models an open workload; usually an open

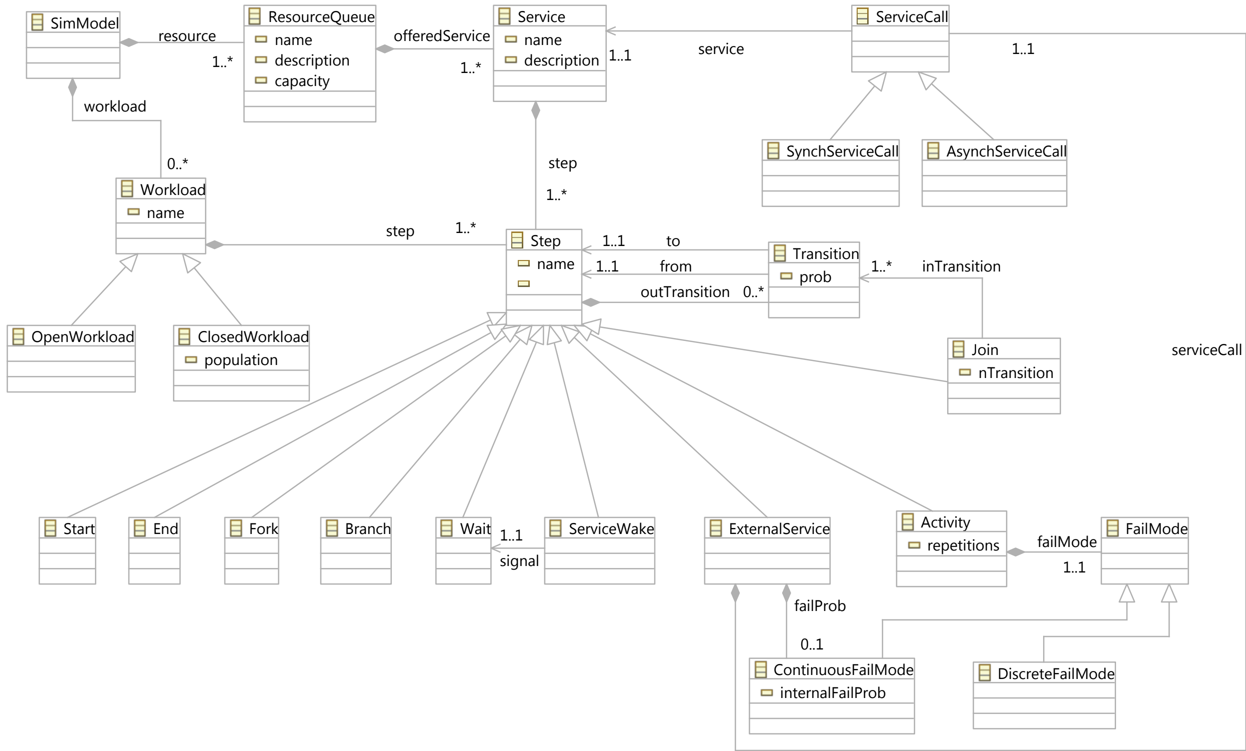

Figure 8.1. SimJava meta model

workload is characterized by the arrival rate of users and this is not an exception, indeed we have the *arrivalProcess* relation whose aim is precisely to define the mean number of users that comes to the system from the outside world.

- **ClosedWorkload**: this meta class represents a closed workload; all the considerations already done for open workload are also valid for closed workloads, but with an exception, for closed workloads we don't consider the arrival rate of users but we need to know the the mean population of users that run around the system because it can affect the number of times actions are accomplished and therefore can influence reliability.

- **ResourceQueue**: this meta class represents a logical or a physical component

of the system. It has the role of requests handler, therefore it is its responsibility to monitor when the capacity (that is the maximum number of parallel requests that can be served) of a resource is full and as soon as there is a free servant to send available requests to it. But it also have to monitor the end of requests to update the number of requests that can be served by a resource.

- **Service**: this meta class represents the concept of a service that a resource can provide to someone that need it. Actually its main role is to be a container of SimJava Steps.

- **Step**: this meta class represents the atomic action of a Service (Services are container of Steps). To provide a Service each Step interact with other Steps using Transitions, indeed the *outTransition* attribute is the set of Transitions outgoing from a Step. There are Steps that are only used to control the execution flow of a Service or a Workload and therefore don't affect the simulation and there are Steps that effectively influence the simulation time and therefore the reliability and the performance of the system.

- **Transition**: this meta class defines a link between two Steps and the probability (the attribute *prob*) that given a Step, the next Step will be the one linked with the Transition.

- **Start**: this is a Step that is used to start a Service or a Workload. It doesn't affect the simulation time.

- **End**: this is a Step that is used to end a Service or a Workload. It doesn't affect the simulation time.

- **Branch**: this Step is used to model branches with a given probability. Obviously the sum of the probabilities of outgoing Transitions has to be equal to 1. It doesn't affect the simulation time.

- **Fork**: this Step is used to model the execution of parallel flows. It is very similar to Branch but in this case all the outgoing Transitions have a probability equal to 1 because all of them are executed at the same time. It doesn't affect the simulation time.

- **Join**: this Step is strictly coupled with the Fork step because Join merges parallel flows originated by a Fork. The attribute *nTransition* defines the number of requests that the step has to wait before doing a request to its next step. It doesn't affect the simulation time.

- **Wait**: this Step models the wait for an event. While waiting, the step is not in execution and the simulation of the related part of the system is completely suspended (while other parts of the system continue their execution), therefore it doesn't affect the simulation time.

- **ServiceWake**: this Step is used to awake the simulation for a Wait step (the Wait step to awake is selected using the attribute *signal*). It doesn't affect the simulation time.

- **FailMode**: mate class that defines the failure mode used for an Activity (see later) or an ExternalService (see later).

- **DiscreteFailMode**: this fail mode defines the time (using a probability distribution function) after which the Activity is considered failed.

- **ContinuousFailMode**: this fail mode defines the probability (between 0.0 and 1.0) that an Activity or an ExternalService have to fail.

- **Activity**: this meta class represents a phase in which the request is served by the resource the activity run on. An Activity can be repeated *repetitions* times, has a failure mode (the attribute *failMode*) and a service time (the attribute *serviceTime* that defines the time required to serve the request). The failure mode strongly influences the way the simulation proceeds for an Activity:

  – failure mode is a DiscreteFailMode[1]: this failure mode models a situation where we have an execution time and a failure time (both of them expressed using stochastic distributions) that "race" between them; if the execution time wins we have a normal execution, while if the failure time wins we have a failure. To better understand this concept imagine that once the service request is received from the probability distribution function *serviceTime* we generate a service time, then from the probability distribution function *internalFailTime* we generate a failure time;

if the failure time is smaller than the service time then the activity is considered failed and we have to update all reliability indexes, otherwise (failure time greater than the service time) we advance the simulation clock of the service time, update statistics and then we can proceed to the next step.

  - failure mode is a ContinuousFailMode[1]: this failure mode models a situation where we know that an activity will fail with a certain probability no matter what is its duration. To better understand this concept imagine that once the service request is received using a uniform distribution we generate a number between 0 and 1, then if this number is smaller than the *internalFailProb* the Activity can be considered failed and therefore we can update statistics and close the Activity; if instead the generated number is greater than the *internalFailProb* attribute we can generate a service time from *serviceTime* and use this value to increment the simulation time and then update statistics before going to the next step.

- **ServiceCall**: this meta class models a service request to another Service.
- **SynchServiceCall**: meta class that represents a synchronous service call, that is a service call that wait for a request before proceeding.
- **AsynchServiceCall**: meta class that represents an asynchronous service call, that is a service call that doesn't wait any response but after doing the request simply goes ahead.
- **ExternalService**: this meta class represents the calling style of a service request; indeed it is a container of a ServiceCall (that can be synchronous or asynchronous) and a ContinuousFailMode[2] (that represents the failure mode of the service call). We have two different behaviors of the meta class according to the service call type chosen for ExternalService:

[1] Understanding the meaning of DiscreteFailMode and ContinuousFailMode the reader could complain that the two names should be swapped to better fit the name of the meta classes with the concepts they model; this is absolutely true but due to some compatibility issues with previous releases of the meta model the names are left as we described here.

[2] The actual meta model considers only ContinuousFailMode as the possible failure mode for an ExternalService. This is a limitation for the meta model that hasn't any technical or conceptual motivation; in the near future also the DiscreteFailMode will be available as failure mode of the ExternalService meta class

– synchronous service call (SynchServiceCall): after receiving the request we generate a random number between 0 and 1, if it is smaller than the *failProb* attribute the step is failed. If instead it is greater than it we can call the service specified by the *service* attribute of the SynchServiceCall meta class; at this point the execution of the current step is suspended until the called service successfully ends with a response or it fails. If the called service ends the requested service without any failure we can go to the step following this ExternalService, otherwise the called service is failed and we have to consider also this ExternalService step as failed.

– asynchronous service call (AsynchServiceCall): after receiving the request we generate a random number between 0 and 1, if it is smaller than the *failProb* attribute the step is failed. If instead it is greater than it we can call the service specified by the *service* attribute of the AsynchServiceCall meta class; this time we don't care about the responses because an asynchronous request simply requests a service and goes ahead. Here we don't consider the *dependsOn* attribute of KLAPER asynchronous calls; currently it is always considered as *false*, but this is a limitation of the KLAPER to SimJava transformation that will be removed in the near future.

8.2 KLAPER to SimJava transformation, concepts

The KLAPER meta model and the SimJava based one are really close; this is obvious if we think that the latter is an implementation of the behavioral semantic of the former, based on the SimJava Java library. This similarity has as a consequence the fact that a lot of elements of the two meta model are directly mapped one to one.

In the remaining part of this section we will see the concepts that lies behind the transformation rules from the KLAPER meta model to the SimJava based one. In these transformation concepts we will use the term "SimJava" to refer to the SimJava based meta model.

- Each KLAPER Workload is transformed into a SimJava Workload, OpenWorkload or ClosedWorkload, depending from the KLAPER specific Workload type considered (KLAPER has only one Workload meta class to represent both

open and closed workloads, while SimJava has two specific sub-meta classes, one for each specific concept).

- Each KLAPER Resource is transformed into a SimJava ResourceQueue. Services of KLAPER Resources are then transformed into Services of SimJava ResourceQueues.
- KLAPER Behaviors haven't a correspondence into SimJava; they are mapped directly to the SimJava Service meta class.
- Each KLAPER Transition is transformed into a SimJava Transition. Both meta classes have the *prob* attribute that expresses the probability that the transition occurs. But while KLAPER Transitions are stored into Behaviors, SimJava stores Transitions into Steps because it lacks the concept of a Behavior.
- Each KLAPER Service is transformed into a SimJava Service.
- Each KLAPER Start step is transformed into a SimJava Start step.
- Each KLAPER End step is transformed into a SimJava End step.
- Each KLAPER Branch step is transformed into a SimJava Branch step.
- Each KLAPER Fork step is transformed into a SimJava Fork step.
- Each KLAPER Join step is transformed into a SimJava Join step. The Join step is the only SimJava step that stores the information about incoming transitions; this is necessary to properly handle the *nTransition* attribute (directly converted from the KLAPER *transitionsNeededToGo* attribute).
- Each KLAPER Wait step is transformed into a SimJava Wait step.
- Each KLAPER Activity is transformed into a SimJava Activity. KLAPER *repetitions* are considered, while nested behaviors are not transformed (this is still under development). If an *internalFailTime* is specified for the KLAPER Activity, the SimJava Activity has a discrete failure mode, otherwise it has a continuous failure mode.

- Each KLAPER ServiceControl that acts like a signal is transformed into a SimJava ServiceWake.

- Each KLPAER ServiceControl that acts like a service call, is transformed into a SimJava ExternalService containing a SimJava SynchServiceCall if the KLAPER ServiceControl is synchronous, or a SimJava AsynchServiceCall if the KLAPER ServiceControl is asynchronous. ExternalServices can have only a continuous fail mode that in the transformation rule is transformed from the *internalFailProb* attribute of the KLAPER ServiceControl meta class if this last one is specified.

After discussing the general concepts, we can now analyze the concrete implementation of transformation rules.

8.3 KLAPER to SimJava transformation, implementation

The QVT transformation from KLAPER to SimJava (as usual expressed using the Xtend tool) is very simple due the fact that the two meta models are very close to each other.

We can start transforming a KLAPER KlaperModel into its corresponding SimJava container with the rule

```
25 // Klaper KlaperModel class
26 private create simulator::core::SimModel this transformModel(klaper::core::
       KlaperModel m):
27     this.setResource(m.resource.transformResource())->
28 //        this.setWorkload(m.workload.select(w|w.type.toString()=='OPEN').
       transformOpenWorkload())->
29         this.workload.addAll(m.workload.select(w|w.type.toString()=='OPEN').
              transformOpenWorkload())->
30 //     this.setWorkload(m.workload.select(w|w.type.toString()=='CLOSED').
       transformClosedWorkload())->
31     this.workload.addAll(m.workload.select(w|w.type.toString()=='CLOSED').
         transformClosedWorkload())->
32     this;
```

where we put into a SimModel meta class all the SimJava Resources (line 27) and Workloads (open workload at line 29 and closed workload at line 31) of the input model.

A SimJava OpenWorkload is created from a KLAPER Workload with the rule

```
34 private create simulator::core::OpenWorkload this transformOpenWorkload(klaper::
       core::Workload w):
35         this.setName(w.name)->
36         this.setStep(w.behavior.step.transformStep())->
37         this.setArrivalProcess(w.arrivalProcess.transformDistribution())->
38         this;
```

where at line 36 we transform all the KLAPER Steps found into the KLAPER Workload in SimJava Steps and then at line 37 we set the arrival process of the open workload. We have to note that even if probability distribution functions are defined in the same way both in KLAPER and in SimJava (probability distribution functions represented in the SimJava based meta model had been directly cloned from the KLAPER meta model), they must be converted in any case, using the transformDistribution() function because even if identical they belong to different meta models; however the transformation is trivial because we have a one to one matching.

SimJava ClosedWorkload are transformed in the same way as OpenWorkload with the rule

```
40 private create simulator::core::ClosedWorkload this transformClosedWorkload(klaper
       ::core::Workload w):
41         this.setPopulation(w.population)->
42         this.setName(w.name)->
43         this.setStep(w.behavior.step.transformStep())->
44         this;
```

where the only change is that in closed workload rather than setting the arrival process we have to set the *population* attribute.

KLAPER Resources are directly transformed into ResourceQueue with the rule

```
46 private create simulator::core::ResourceQueue this transformResource(klaper::core::
       Resource r):
47         this.setCapacity(r.capacity.toInteger())->
48         this.setDescription(r.description)->
49         this.setName(r.name)->
50         this.setOfferedService(r.offeredService.transformService())->
51         this;
```

where we can see that the mapping between the KLAPER meta model and the SimJava meta model is really one to one.

A KLAPER Service is transformed into a SimJava Service using the rule

```
53 private create simulator::core::Service this transformService(klaper::core::Service
        s):
```

```
54          this.setName(s.name)->
55          this.setDescription(s.description)->
56          this.setStep(s.behavior.step.transformStep())->
57          this;
```

where we can see one of the few differences between KLAPER and SimJava, SimJava doesn't have the concept of behavior; in SimJava the KLAPER concepts of Service and Behavior are merged into the Service meta class. At line 56 we can see how KLAPER Steps are transformed into SimJava Steps and then added to a SimJava Service.

The first step of a KLAPER Behavior is always the Start step whose transformation rule is

```
62  private create simulator::core::Start this transformStep(klaper::core::Start s):
63          this.setName(s.name)->
64          this.setOutTransition(((klaper::core::Behavior)s.eContainer).transition.
                select(t|t.from.name==s.name).transformTransition())->
65          this;
```

where at line 64 we select all KLAPER Transitions that have the Start step as source (see the condition $select(t|t.from.name == s.name)$) and then we transform them into SimJava Transitions belongings from (because they are contained into) the SimJava Start step.

The KLAPER End step is trivial to transform to its corresponding SimJava Step; its transformation rule is

```
67  private create simulator::core::End this transformStep(klaper::core::End e):
68          this.setName(e.name)->
69  //      this.setOutTransition(((klaper::core::Behavior)e.eContainer).transition.
        select(t|t.from.name==e.name).transformTransition())->
70          this;
```

where the only thing we have to set is the name of the Step (see line 68), because the End step in SimJava has no any following step and therefore there is nothing else to set.

Transformation rules for KLAPER Branch step and KLAPER Fork step are identical

```
72  private create simulator::core::Branch this transformStep(klaper::core::Branch b):
73          this.setName(b.name)->
74          this.setOutTransition(((klaper::core::Behavior)b.eContainer).transition.
                select(t|t.from.name==b.name).transformTransition())->
75          this;
76
77  private create simulator::core::Fork this transformStep(klaper::core::Fork f):
```

```
78         this.setName(f.name)->
79         this.setOutTransition(((klaper::core::Behavior)f.eContainer).transition.
               select(t|t.from.name==f.name).transformTransition())->
80         this;
```

both the transformation rules set the name of the Step (lines 73 and 78) and the outgoing Transitions (lines 74 and 79).

The transformation rule for the KLAPER Join step is similar to other rules seen so far with a little difference

```
82 private create simulator::core::Join this transformStep(klaper::core::Join j):
83         this.setName(j.name)->
84         this.setNTransition(j.transitionsNeededToGo)->
85         this.setInTransition(((klaper::core::Behavior)j.eContainer).transition.
               select(t|t.to.name==j.name).transformTransition())->
86         this.setOutTransition(((klaper::core::Behavior)j.eContainer).transition.
               select(t|t.from.name==j.name).transformTransition())->
87         this;
```

where the only new thing we have to analyze is at line 85; here we set a list of incoming Transitions (selected using a way very similar to that used for outgoing transition) that will be used during the M2T transformation for a comparison with the *nTransition* attribute to decide when the Java code representing a Join step can resume its execution.

The KLAPER Wait step has the same transformation rule already seen for other Steps

```
89 private create simulator::core::Wait this transformStep(klaper::core::Wait w):
90         this.setName(w.name)->
91         this.setOutTransition(((klaper::core::Behavior)w.eContainer).transition.
               select(t|t.from.name==w.name).transformTransition())->
92         this;
```

we already saw identical transformation rules so there is no need to add anything anymore.

The transformation rule for the KLAPER Activity step is something more interesting if compared to other Steps

```
94 private create simulator::core::Activity this transformStep(klaper::core::Activity
       a):
95         this.setName(a.name)->
96         this.setOutTransition(((klaper::core::Behavior)a.eContainer).transition.
               select(t|t.from.name==a.name).transformTransition())->
97         this.setRepetitions(a.repetitions.getMean().toInteger())->
98         this.setServiceTime(a.internalExecTime.transformDistribution())->
99         a.internalFailTime!=null?
```

```
100                 this.setFailMode(a.internalFailTime.transformDiscreteFailMode()):
                        this.setFailMode(a.internalFailProb.transformContinuousFailMode
                        ())->
101         this;
```

where at line 95, 96 and 97 we only set attributes. At line 98 we set the SimJava *serviceTime* directly from the KLAPER *internalExecTime*; then if an *internalFailTime* has been specified for the KLAPER Activity we set a DiscreteFailMode for the SimJava Activity, otherwise we set a ContinuousFailMode for the SimJava Activity.

KLAPER ServiceControl meta class has two different transformation rules depending on the service call or the signal nature of the meta class itself. The transformation rule used to represent a service call is

```
103 private create simulator::core::ExternalService this transformStep(klaper::core::
        ServiceControl sc):
104         sc.binding.call!=null?
105         (
106         this.setName(sc.name)->
107         this.setOutTransition(((klaper::core::Behavior)sc.eContainer).transition.
                select(t|t.from.name==sc.name).transformTransition())->
108         sc.internalFailProb!=null?this.setFailProb(sc.internalFailProb.
                transformContinuousFailMode()):this.setFailProb(null)->
109         sc.isSynch==true?
110                 this.setServiceCall(sc.transformSynchServiceCall()):this.
                        setServiceCall(sc.transformAsynchServiceCall())->
111         this)
112         :null;
```

where at line 104 we check the service call nature of ServiceCall verifying if the KLAPER Binding meta class instance references some KLAPER Service. If so we can set the name of the LQN ExternalService (line 106) and the set of outgoing LQN Transitions (line 107). Then at line 108 we set the failure mode (if present), where we recall that ExternalServices can have only a ContinuousFialMode (while Activities can also have a DiscreteFailMode); if the KLAPER ServiceCall has no *internalFailProb* specified, then the LQN failure mode is not set. Before closing the rule we only need to set the right service call type checking the KLAPER *isSynch* attribute (see line 109), in case its value is *true* we instantiate an LQN SynchServiceCall, otherwise we instantiate an LQN AsynchServiceCall (see line 110). The final null of line 112 is only due to an Xtend limitation that doesn't have an "if" construct but presents only an "if then else" construct where the "else" condition has to be set to null to realize a simple "if" condition.

The transformation rule for the KALPER ServiceControl in its meaning of signal

is very close to the one used for service call but with some differences

```
114 private create simulator::core::ServiceWake this transformStep(klaper::core::
        ServiceControl sc):
115         sc.binding.signal!=null?
116         (
117         this.setName(sc.name)->
118         this.setOutTransition(((klaper::core::Behavior)sc.eContainer).transition.
                select(t|t.from.name==sc.name).transformTransition())->
119         this.setSignal(sc.binding.signal.transformStep())->
120         this
121         )
122         :null;
```

where the main difference is that this time we create a ServiceWake step rather than an ExternalService. Then we have to consider that signals can't fail, because they are instantaneus, and don't have a synchronization, they are asynchronous by definition; therefore in this case we don't have to set a failure mode nor a synchronization mode, we only need to set the LQN Wait step to awake (see line 119).

All the LQN Steps created by the transformation rules seen so far are linked together using LQN Transitions whose transformation rule is

```
165 private create simulator::core::Transition this transformTransition(klaper::core::
        Transition t):
166         this.setFrom(t.from.transformStep())->
167         this.setTo(t.to.transformStep())->
168         this.setProb((t.prob>0 && t.prob<=1)?t.prob:1.0)->
169         this;
```

where we have a one to one correspondence, indeed LQN source Step (the *from* attribute) is directly taken from the related KLAPER source step (see line 166), LQN target Step (the *to* attribute) is directly taken from the related KLAPER target step (see line 167) and the LQN Transition probability is converted from the KLAPER Transition probability if it is specified, otherwise it is set to the default 1.0.

8.4 Using the SimJava meta model

We already said that SimJava is not an analytical solver like those we use with DTMC or LQN, but it is a simulation tool in the sense that it provides a Java library we can use to build a system that, even if it is not the real system in terms of functional requirements, has a behavior really close the the real system behavior in terms of performance and reliability.

While with other models we generate an input file to be provided as input to some sort of solver, with SimJava we generate Java code that uses some specific APIs provided by the SimJava Java library (available as a jar package). This means that the result of the KLAPER tool execution this time is pure Java code; we need a Java compiler to build the generated code into an application that, once run, returns as output a set of measures collected during the simulation.

8.4.1 Tool limitations

The SimJava based meta model was the first developed inside the KLAPER project and therefore it suffers from some lack of experience. This has led to some limitations due to both the structure of the used meta model and the implementation of the model-to-text transformation.

The current tool presents the following limitations (they are not limitation of SimJava itself, but only of the current implementation of the tool):

- Limitations due to the meta model structure:
    1. Acquire and Release steps are not considered.
    2. Bindings cannot be reconfigured.
    3. KLAPER nested behaviors cannot be represented; there isn't any meta model entity that permits the mapping of such a concept.
    4. All the service requests are sotisfied using a FIFO policy; there isn't any other scheduling policy available.
- Limitations due to the model-to-text transformation implementation:
    5. FormalParam and ActualParam are not considered during the transformation.
    6. Every SimJava Step has been implemented extending a Java Thread, this means that even for systems not so complex we can easily reach the maximum number of available threads of the Java virtual machine.

We have to note that limitations 4 and 5 are the reason why actually the SimJava based meta model tool is not usefull to analyze performance. In addition, the meta model is too much close to the KLAPER meta model rather than to what should

have been a real SimJava meta model; this lead to a model-to-text transformation more complex than expected.

These meta model and transformation limitations obviously restrict the range of input models that can be successfully analyzed by the tool. Indeed with the current SimJava based meta model and the current implementation of the model-to-text transformation we can only simulate very simple systems (this limitation obviously is not present for LQN and DTMC that use a completely independent environment). In any case due to some space requirements and we can't deeper analyze these limitations; however all the needed information (including an extensive discussion of the Java code generation) can be found in [9].

Chapter 9

Transforming to LQN

While DTMC and the SimJava based simulator are models for reliability evaluation (actually SimJava is also capable of performance evaluation but the use of this feature in KLAPER is still under development), LQN (Layered Queueing Network, see section 3.3 for more details about the theoretical aspects of LQN) is a model whose aim is to evaluate performance, therefore into the transformation from KLAPER to LQN we will find some elements of the KLAPER meta model that we haven't used so far.

9.1 The LQN meta model

KLAPER relies on a model driven approach and to do so it needs meta models of the quality analysis methodology we want to apply. LQN creators don't provide a meta model, so to solve this problem we decided to build the meta model ourself (see figure 9.1). We chose to use as starting point the xml schema (see [31]) used by input files of lqns, that is the analytical solver developed by the Carleton University of Ottawa (home of the guys who developed LQN). The use of a meta model built in this way guarantee to us two very important things: the first is that the meta model should be semantically correct because it derives directly from a work done by the creators of LQN, the second is that in this way the M2T transformation is very easy to write because there is a one-to-one correspondence between the meta model and the final code to generate (that must be compliant with the xsd adopted by the lqns tool).

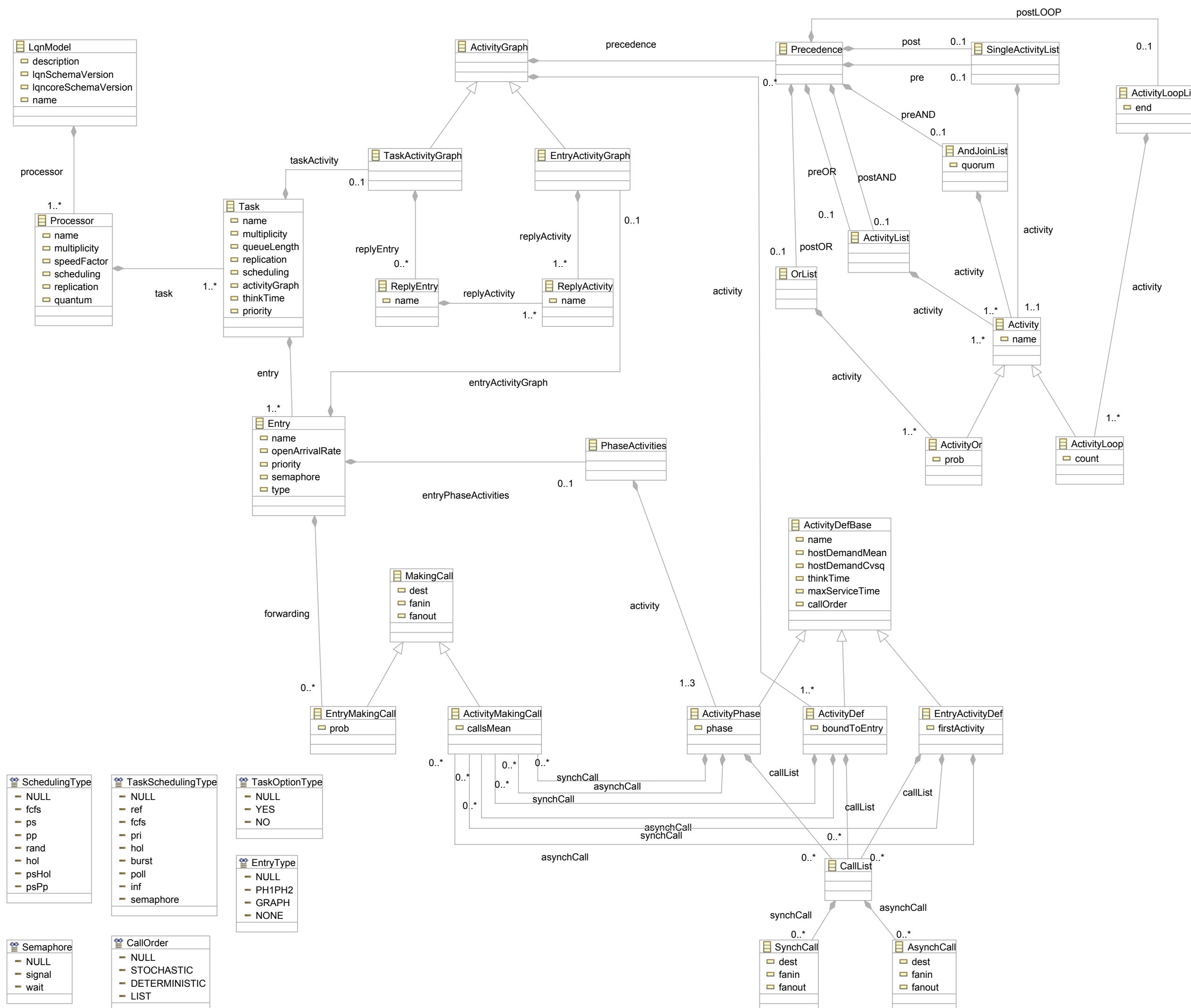

Figure 9.1. LQN meta model

Here we will present an overview of the meta classes that compose the meta model; to have more details see chapter 3 of [23].

The LQN meta model consists of the following meta classes:

- **LqnModel**: this is the container of other meta classes; it also contains some additional information about the model like the reference xsd schema and the description of the model itself.

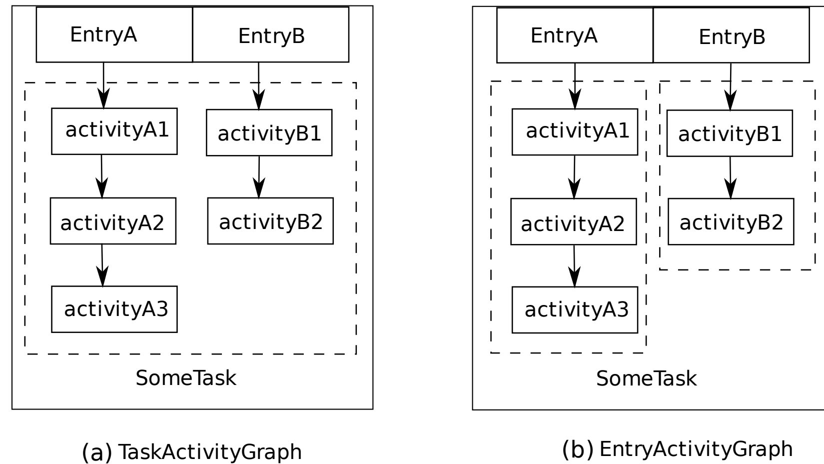

Figure 9.2. LQN representation of TaskActivityGraph (a) and EntryActivityGraph (b))

- **Processor**: this meta class represents an LQN processor and stores information about the processor such as multiplicity and replication (already discussed in section 3.3), scheduling policy and the quantum (the interval of time each task is executed when the selected scheduling policy is a round robin like policy).

- **Task**: represents the LQN concept of a task and stores all the related information.

- **Entry**: this meta class represents the LQN concept of an entry with all related attributes.

- **TaskActivityGraph**: in LQN activities can be organized in some different ways into a task; one of this way is to use some activity graphs to define the sequence of activities. This meta class represents exactly this idea and can be considered as a container of activities, with the addition of some information.

- **EntryActivityGraph**: this meta class is another way to represents the graph of the activities within a task, the difference with TaskActivityGraph is that EntryActivityGraph stores task activities grouping them by the activities that refers to a particular Entry while TaskActivityGraph stores all the activities

of a Task. See figure 9.2 to better understand the differences between the two kind of graphs.

- **ActivityGraph**: this is the parent meta class for the TaskActivityGraph and EntryActivityGraph meta classes and describes the common element needed to realize a graph of LQN activities.
- **ReplyActivity**: each Entry of a task returns some kind of service and we already saw that this service is realized by a sequence of activities that can be expressed using TaskActivityGraphs or EntryActivityGraphs; but we need a way to define which Activity effectively replies to a particular Entry and this is the role of the ReplyActivity meta class.
- **ReplyEntry**: this meta class defines the Entry to which an activity replies when the activities graph is expressed using a TaskActivityGraph.
- **PhaseActivities**: this is another way to define the activities accomplished by an Entry; activities can be specified using phases (theoretically we can have as many phases as we want but in practice lqns support up to three phases where the first is the one used to reply to synchronous calls) and these phases are contained into an instance of the PhaseActivities meta class.
- **ActivityDefBase**: this meta class represents the activity concept as an action to do. This is only the main concept that then is specialized according to the context where it is used.
- **ActivityPhase**: it is the implementation of an activity when the activities of an Entry are expressed using phases.
- **ActivityDef**: it is the implementation of an activity when the activities of an Entry are expressed using an activities graph (whether it is a TaskActivityGraph or an EntryActivityGraph).
- **EntryActivityDef**: actually not used and reported into the meta model only because it is present into the lqns xsd schema.
- **CallList**: this meta class is one of the possible way used by activities to make service calls, but even if the lqns solver supports this concept actually we use another mechanism to do calls.

- **SynchCall**: this is a specialization of CallList used for synchronous calls.

- **AsynchCall**: this is a specialization of CallList used for asynchronous calls.

- **MakingCall**: this meta class is the second mechanism used by the lqns solver to make service calls and it is the one actually used by the KLAPER to LQN transformation; it contains concepts like a reference to the called service (the attribute *name*), the number of replicated servers that a client calls (the attribute *fanout*) and the number of replicated clients that call a server (the attribute *fanin*).

- **EntryMakingCall**: this meta class represents the concept of an Entry making a call without specifying any activity, that is a forwarding request; actually this is not used by the KLAPER to LQN transformation even if lqns can handle it.

- **ActivityMakingCall**: represents an activity that makes a synchronous or an asynchronous call specifying a mean number of calls (using the attribute *callsMean*); this is the way calls are done with the KLAPER to LQN transformation.

- **Precedence**: this meta class is used to specify the way activities are linked between them to build an activities graph; a Precedence always have one or more “pre” (or AND, that comes before) elements followed by one or more “post” (or OR, that comes after) elements.

- **Activity**: this meta class represents the reference to an activity (identified by the attribute *name*).

- **ORList**: this meta class is a precedence used to describe branch conditions;

- **ActivityOR**: this meta class represents a single branch of a branch condition with the probability (the attribute *prob*) associated to the related branch.

- **ActivityList**: when used as a “pre” precedence this meta class represents the merging of different branches previously started from a branch condition, while when used as a “post” precedence it represents a typical fork condition for multiple concurrent activities.

- **AndJoinList**: this meta class is a precedence used to represent join conditions, that is a synchronization point for concurrent activities.
- **SingleActivityList**: this class used as “pre” or “post” precedence always represents a single activity into a simple activities sequence.
- **ActivityLoopList**: this is the precedence used to represent loops, where the *end* attribute denotes the activity executed at the end of the loop.
- **ActivityLoop**: this meta class define a reference to an activity that is repeated *count* times before going out the loop.

9.2 KLAPER to LQN transformation, concepts

Now that we know how the LQN meta model is defined, before going deep into the details of the transformation rules with the related code, in this section we will see the general concepts that we apply to the transformation from the KLAPER meta model to the LQN meta model:

- Each KLAPER Workload is mapped into a special LQN Task that is called *reference task*. The peculiarity of a reference Task is that it has an Entry but this Entry can never be called by anyone. A closed workload is represented setting the multiplicity of the LQN Task to the population number of the KLAPER Workload, while an open workload is represented setting the arrival rate of the Task's Entry. The mapping also creates a special processor reserved to the reference Task execution not affecting other real Processors execution.
- KLAPER doesn't make difference between hardware and software resources; they are all represented as KLAPER Resources. LQN maps hardware resource to Processors plus the related base service implemented as a simple Task (with a single Entry), while software resources are mapped as common LQN Tasks. Actually hardware resources are recognized using the *type* attribute of KLAPER Resources; the only values currently accepted for hardware resources are *cpu*, *network* and *disk*. We will see in the next section some examples of hardware resources transformation.
- Each KLAPER Service is mapped into an LQN Entry of a Task.

- Each KLAPER Behavior is mapped into an LQN TaskActivityGraph.
- LQN EntryActivityGraphs are not used by the transformation rules.
- Each KLAPER Start step is mapped into an LQN Activity with a service time equal to 0.0 and a simple Precedence.
- Each KLAPER End step is mapped into an LQN Activity with a service time equal to 0.0 and a simple Precedence.
- Each KLAPER Wait step is mapped into an LQN Activity with a service time equal to 0.0 and a simple Precedence.
- Each KLAPER Branch step is mapped into an LQN Activity with a service time equal to 0.0 and an LQN PostOR precedence.
- Each KLAPER Fork step is mapped into an LQN Activity with a service time equal to 0.0 and an LQN PostAND precedence.
- Each KLAPER Join step is mapped into an LQN Activity with a service time equal to 0.0 and an LQN PreAnd precedence.
- KLAPER Activities are mapped in different ways:
  - if the KLAPER Activity has no nested behaviors and the repetition number is equal to 1 then the activity is mapped into a simple LQN Activity with the related service time.
  - if the KLAPER Activity has a repetition number greater than 1 then the activity is mapped into an LQN Activity with a loop Precedence.
  - if the KLAPER Activity has a nested behavior the activity is mapped into an LQN Activity with a loop Precedence (the loop counter is set to 1 if we have a simple nested behavior or to the mean number of KLAPER Activity repetitions if this attribute is specified); the nested behavior is then mapped like normal KLAPER Behaviors but using as root the LQN loop Activity.
- KLAPER ServiceControl are mapped just like KLAPER Activities into LQN Activities, but with the only difference of service calls. The mapping rule

creates a new LQN ActivityMakingCall instance that is added to the synchCall list of the LQN Activity if the KLAPER *isSynch* attribute is equal to true, otherwise the new instance is added to the asynchCall list of the LQN Activity.

This is only an overview of general concepts applied to the transformation rules between the KLAPER meta model and the LQN meta model. In the following section we will better analyze each of these rules with examples and the references to the QVT code used into the real implementation of KLAPER tools.

9.3 KLAPER to LQN transformation, implementation

As already done with the other meta models we will present the QVT transformation rules from KLAPER to LQN using the Xtend language.

One of the main problems transforming from KLAPER to LQN is that, as already said, KLAPER doesn't make difference between hardware and software resources, they are considered exactly at the same level, while LQN is strongly based on this separation due to the difference between the concepts of processors and tasks. In LQN every task runs over a processor and therefore we need to know the deployment of the system to correctly transform from KLAPER. The solution we choose to overcome this problem is to split the transformation process in two steps: in the first step we map all KLAPER resources to LQN tasks and we allocate them (both hardware and software resources) into a dummy processor, then in the second step we reconfigure the system with the correct deployment moving each task to its own processor[1]. This is necessary because when we analyze the KLAPER model for the first time we don't yet know the hardware configuration of the system that is later computed analyzing the type of KLAPER resources, but we will see better later how the deployment identification works. All that we said until now is visible into the following excerpt of code

```
28 // starting point for the transformation
29 // (this extension is called from the workflow!)
30 Object klaper2lqn(klaper::core::KlaperModel m):
31 //       m.transformModel();
32          m.transformationSteps();
```

[1]at the end of this process the dummy processor is removed from the model

```
33
34
35 private lqn::LqnModel transformationSteps(klaper::core::KlaperModel m):
36         m.transformationStep1().transformationStep2(m);
```

After that starts the first part of the transformation with the transformationStep1() function

```
40 private lqn::LqnModel transformationStep1(klaper::core::KlaperModel m):
41         reportWarning("Model transformation running") ->
42         m.transformModel();
```

that simply starts the instantiation of LQN meta classes.

The first action to do is to create a container for all LQN Processors, but for now we will put into this container only a single dummy processor plus some other specific processors (those needed by workloads)

```
56 /**
57  * creates the initial model with a dummy processor
58  */
59 private create lqn::LqnModel this transformModel(klaper::core::KlaperModel m):
60         this.setName("LqnModel") ->
61         this.setDescription("Model auto-genereted by Klaper tool") ->
62         this.setLqncoreSchemaVersion("1.0") ->
63         this.setLqnSchemaVersion("1.0") ->
64         this.processor.add(createDummyProcessor(m.resource)) ->         // creates
               a dummy processor
65         this.processor.addAll(m.workload.transformWorkload()) ->
66         this;
```

where on line 64 we create the dummy processor and on line 65 we create a single processor for every workload present into the model.

Workloads are very important for performance evaluation, because modifying workloads we can greatly change the behavior of the system from the point of view of performances; therefore the first part of the KALPER to LQN transformation is completely dedicated to how a KLAPER Workload meta class can be represented in LQN.

LQN doesn't have a dedicated concept to represent workloads, so we have to map them to the structure of Processors and Tasks; more precisely every KLAPER Workload is transformed into a pair composed by an LQN Processor and an LQN Task that runs over it (see figure 9.3). To represent workloads LQN relies on special Tasks, called *reference tasks*; they cannot be called by anyone but to be triggered they rely on external requests represented by requests inter-arrival rate (expressed using Entries) for open workloads or on a fixed number of users that live into the

system represented using a population number (expressed using the multiplicity of the related Task) for closed workloads.

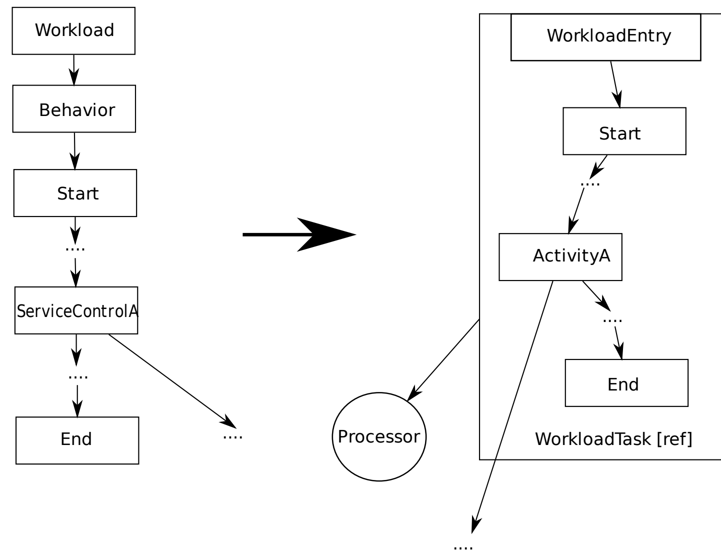

Figure 9.3. Transformation of a KLAPER (open or closed) Workload into an LQN Processor and a reference Task

```
81 /**
82  * transform a workload into a processor + task
83  */
84 private create lqn::Processor this transformWorkload(klaper::core::Workload w):
85         this.setName(w.name) ->
86         this.setMultiplicity(1) ->
87         this.setReplication(1) ->
88         this.setScheduling(lqn::SchedulingType::ps) ->
89         this.setQuantum(0.1 * 0.1 * 0.1 * 0.1 * 0.1) -> // only because 0.00001
               give a parse error! but why???
90         switch(w.type.toString())
91         {
92                 case klaper::core::WorkloadType::OPEN.toString():
93                         this.task.add(w.createOpenWorkloadTask())
94                 case klaper::core::WorkloadType::CLOSED.toString():
95                         this.task.add(w.createClosedWorkloadTask())
96                 default:
97                         reportError("For workload '" + w.name + "' unknown type '"
                               + w.type.toString() + "'")
98         } ->
99         this;
```

where from line 85 to 89 we set the newly created Processor's attributes, while from lines 90 to 98 we create a Task that represents the actions done into the Workload. The switch control over a string on line 90 is simply due to an Xtend bug that sometimes is not capable of applying the switch construct over an enumeration; then we create Tasks of different type depending on the open or closed nature of the workload.

For an open Workload the transformation rules are

```
102 /**
103  * creates a task rappresenting an open workload
104  */
105 private create lqn::Task this createOpenWorkloadTask(klaper::core::Workload w):
106         this.setName(w.name) ->
107         this.setMultiplicity(32700) ->
108         this.setReplication(1) ->
109         this.setScheduling(lqn::TaskSchedulingType::fcfs) ->
110         this.setActivityGraph(lqn::TaskOptionType::YES) ->
111         this.entry.add(w.createOpenWorkloadEntry()) ->
112         this.setTaskActivity(w.createTaskActivityGraph(w.createOpenWorkloadEntry())
                ) ->
113         this;
114
115
116 /**
117  * creates an Entry that sets requests at an open arrival rate for an open workload
118  */
119 private create lqn::Entry this createOpenWorkloadEntry(klaper::core::Workload w):
120         this.setName(w.name.toLowerCase()) ->
121         this.setType(lqn::EntryType::NONE) ->
122         this.setOpenArrivalRate(w.arrivalProcess.getMean()) ->
123         this;
```

where first on line 109 we create a new Task with an "fcfs" scheduling policy (that is a FIFO scheduling policy), then on line 110 we say that we will use an activities graph setting the *activityGraph* attribute to yes and to finish we add an Entry and a TaskActivityGraph (lines 111 and 112) to the new Task. The createTaskActivityGraph() will be analyzed later because it is the same used for regular Tasks. Instead the createOpenWorkloadEntry() is presented from line 116 to line 123 where we create a simple LQN Entry with an open arrival rate (typical for open workloads) computed from the arrival process of the KLAPER Workload.

One thing to note is that in KLAPER we always deal with probability distribution functions to be as general as possible, while in LQN the only thing we care about distribution function is the related mean that obviously we have to compute.

For a closed Workload the transformation rules are

```
126 /**
127  * creates a task rappresenting a closed workload
128  */
129 private create lqn::Task this createClosedWorkloadTask(klaper::core::Workload w):
130         this.setName(w.name) ->
131         this.setMultiplicity(w.population) ->
132         this.setReplication(1) ->
133         this.setScheduling(lqn::TaskSchedulingType::ref) ->
134         this.setActivityGraph(lqn::TaskOptionType::YES) ->
135         this.entry.add(w.createClosedWorkloadEntry()) ->
136         this.setTaskActivity(w.createTaskActivityGraph(w.createClosedWorkloadEntry
                ())) ->
137         this;
138 
139 /**
140  * creates an Entry that sets requests for a closed workload
141  */
142 private create lqn::Entry this createClosedWorkloadEntry(klaper::core::Workload w):
143         this.setName(w.name.toLowerCase()) ->
144         this.setType(lqn::EntryType::NONE) ->
145         this;
```

where the applied transformations are quite similar to those used for open workloads with three simple differencies: on line 131 the multiplicity of the task is set to the population number of the closed workload instead of a really big number used for open workloads to represent the infinite value, on line 133 the scheduling policy is set to “ref” to represent a reference task (the typical task used for closed workloads) and finally from line 142 to line 145 the Task’s Entry doesn’t have any open arrival rate because it is not needed in closed workloads.

After transforming Wokloads, the next step is to transform all the Resources of the model; this is done with the following rule

```
148 /**
149  * creates an lqn Task
150  */
151 private create lqn::Task this transformTask(klaper::core::Resource r):
152         this.setName(r.name) ->
153         this.setMultiplicity(r.capacity.toInteger()) ->
154         this.setReplication(1) ->
155         this.setScheduling(r.schedulingPolicy.toLqnScheduling()) ->
156         this.setActivityGraph(lqn::TaskOptionType::YES) ->
157         this.entry.addAll(r.offeredService.createEntry()) ->
158         this.setTaskActivity(r.createTaskActivityGraph()) ->
159         this;
```

in our KLAPER to LQN transformation we use the method of LQN TaskActivityGraphs to convert all KLAPER Services and related steps so, after setting some attributes, on line 156 we declare that we want to use TaskActivityGraph as steps container (attribute *activityGraph* set to *yes*), on line 157 we transform all the Entries of the current Task and then on line 158 we create the TaskActivityGraph.

Each Task has one or more Entries and each Entry is transformed using the simple rule

```
162 /**
163  * creates an Lqn Entry for regular tasks
164  */
165 private create lqn::Entry this createEntry(klaper::core::Service s):
166         this.setName(s.name) ->
167         this.setType(lqn::EntryType::NONE) ->
168         this;
```

where we set the name and the type of the Entry. The *type* attribute set to *NONE* is the way the LQN meta model uses to define Entries when a TaskActivityGraph is used (the attribute has a different value for EntryActivityGraphs and for phases).

At this point we only need to set TaskActivityGraph; but we need two different rules depending on the subject of the transformation. If we are generating a TaskActivityGraph from a Workload we use the rule

```
171 /**
172  * creates an Lqn TaskActivityGraph for workload's behavior
173  */
174 private create lqn::TaskActivityGraph this createTaskActivityGraph(klaper::core::
        Workload w, lqn::Entry entry):
175         let activities = {}:
176         let precedences = {}:
177         w.behavior.buildActivitiesGraph(activities, precedences, entry) ->
178         this.setActivity(activities) ->
179         this.setPrecedence(precedences) ->
180         this.replyEntry.add(w.behavior.step.typeSelect(klaper::core::End).first().
            createReplyEntry("reply_" + w.name)) ->
181         this;
```

that is identical to that we will see later for Resources, except for the *replyEntry* attribute that is directly set to a specific value (line 180) because we are sure that Workloads have only one Entry, while Resources can present more the one single Entry and therefore more than one single *replyEntry* value. Instead if we are generating a TaskActivityGraph from a Resource we have to use the rule

```
184 /**
185  * creates an lqn TaskActivityGraph
```

```
186  */
187 private create lqn::TaskActivityGraph this createTaskActivityGraph(klaper::core::
        Resource r):
188         let activities = {}:
189         let precedences = {}:
190         let reply_entries = {}:
191         r.offeredService.buildActivitiesGraph(activities, precedences,
                reply_entries) ->
192         this.setActivity(activities) ->
193         this.setPrecedence(precedences) ->
194         this.setReplyEntry(reply_entries) ->
195         this;
```

where on lines 192, 193 and 194 we set respectively the Activities of the graph, the Precedences of the graph and the replyEntrys of the graph (this time replyEntry can be more than one and this is why we have to implement two different rules for Workloads and for Resources). The buildActivitiesGraph() function take as input an empty activities list, an empty precedences list and an empty replyEntry list and parses a Behavior to fill these lists.

The buildActivitiesGraph function is very useful because it visits the Behavior activities graph of a KLAPER Service with the rule

```
198 /**
199  * converts a klaper service into a sub-graph of a TaskActivityGraph
200  */
201 private buildActivitiesGraph(klaper::core::Service s, List[ActivityDef] activities,
         List[Precedence] precedences, List[ReplyEntry] reply_entries):
202         s.behavior.buildActivitiesGraph(activities, precedences, s.createEntry())
                ->
203         reply_entries.add(s.behavior.step.typeSelect(klaper::core::End).first().
                createReplyEntry("reply_" + s.name));
```

where on line 202 we truely visit the service Behavior graph and on line 203 we set the replyEntry related to the current service (and therefore related to a specific Entry that represents the current Service).

The ReplyEntries are linked to the concept of the KLAPER End step because they represent the end of an activities flow, therefore it is the moment when a Service can reply to its clients. The rules used to transform an End step into a ReplyActivity are very simple

```
206 /**
207  * creates an Lqn ReplyEntry (for TaskActivityGraph)
208  */
209 private create lqn::ReplyEntry this createReplyEntry(klaper::core::End e, String
        reply_name):
210         this.setName(reply_name) ->
```

```
211         this.replyActivity.add(e.createReplyActivity()) ->
212         this;
213
214
215 /**
216  * creates an Lqn ReplyActivity (for TaskActivityGraph)
217  */
218 private create lqn::ReplyActivity this createReplyActivity(klaper::core::End e):
219         this.setName(e.name) ->
220         this;
```

where we set the name of the activity (the End step) that replies to a specific Entry (that means to a specific service request).

Until now we have transformed the statical part of KLAPER, the part that concerns the infrastructure of the system modeled. Now we have to deal with KLAPER Behaviors and therefore we have to deal with the dynamic (or behavioral) part of KLAPER: Steps.

To start transforming a Behavior with its Steps we need to use the rule

```
223 /**
224  * builds an activity graph (for behaviors and resources)
225  */
226 private buildActivitiesGraph(klaper::core::Behavior b, List[ActivityDef] activities
        , List[Precedence] precedences, lqn::Entry entry):
227         b.step.typeSelect(klaper::core::Start).transformStep(activities,
                precedences, entry) ->
228         null;
```

where we select as starting point of the transformation the first step of each Behavior, the Start step.

The Start step transformation rule is quite simple, considering that the role of this step is only to start a new graph of activities:

```
262 /**
263  * transforms a Start step
264  */
265 private transformStep(klaper::core::Start start, List[ActivityDef] activities, List
        [Precedence] precedences, lqn::Entry entry):
266         let activity = new lqn::ActivityDef:
267         activity.setName(start.name) ->
268         activity.setHostDemandMean(0.0) ->
269         activity.setHostDemandCvsq(1.0) ->
270         activity.setThinkTime(0.0) ->
271         activity.setMaxServiceTime(0.0) ->
272         activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
273         if(entry != null) then // entry can be null for nested behaviors
274         (
275                 activity.setBoundToEntry(entry.name)
```

```
276         ) ->
277         activities.add(activity) ->
278         if(start.out.first().to.in.size == 1) then
279         (
280                 let precedence = new lqn::Precedence:
281                 precedence.setPre(start.createSingleActivityList()) ->
282                 precedence.setPost(start.out.first().to.createSingleActivityList())
                        ->
283                 precedences.add(precedence)
284         ) ->
285         start.out.first().to.transformStep(activities, precedences, null);
```

where from line 243 to line 249 we create a new ActivityDef instance, that represents the current step, and then we set all its attributes (note the *hostDemandMean* set to 0.0, that means that this step doesn't consume any processing resource). Then if the Start step belongs to a normal Behavior (if condition of line 250 equal to true) we link its corresponding activity to the related LQN Entry, instead if the step belongs to a nested behavior (if condition of line 250 equal to false) the linking is not done. At this point the newly created activity is ready to be added to the activities list of a TaskActivityGraph (see line 254). Before ending this transformation rule we check the number of incoming Transitions of the Step that comes after the Start step (see line 255), if it is equal to 1 (and this is always true because Start steps are always the single starting point of a KLAPER Behavior) we have to create a new Precedence (see line 257); this Precedence is composed by a single activity as pre-element (the activity we just created transforming from the Start step, see line 258) and by a single activity as post-element (the activity created by the transformation of the Step on the opposite side of the Transition starting from Start, see line 259).

To close a KLAPER Behavior started with a Start step we need a transformation rule for its counterpart, the End step

```
265 /**
266  * transforms an End step
267  */
268 private transformStep(klaper::core::End end, List[ActivityDef] activities, List[
        Precedence] precedences, lqn::Entry entry):
269         if(!(activities.exists(a|a.name == end.name))) then // the step is
                transformed only once
270         (
271                 let activity = new lqn::ActivityDef:
272                 if(end.in.size > 1) then // for (incoming) OR-join precedence
273                 (
274                         let precedence = new lqn::Precedence:
275                         precedence.setPreOR(end.createActivityListIn()) ->
```

```
276                          precedence.setPost(end.in.first().to.
                                 createSingleActivityList()) ->
277                          precedences.add(precedence)
278                  ) ->
279                  activity.setName(end.name) ->
280                  activity.setHostDemandMean(0.0) ->
281                  activity.setHostDemandCvsq(1.0) ->
282                  activity.setThinkTime(0.0) ->
283                  activity.setMaxServiceTime(0.0) ->
284                  activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
285                  activities.add(activity)
286          );
```

on line 269 we check if the step has been already transformed (this shouldn't never happen for the End step, but it is better to check because we could have an erroneous input model). If the step has never been transformed we can create a new ActivityDef instance (see line 271) and set all its attributes (from line 279 to line 285 and note once again that the *hostDemandMean* attribute is set to 0 because also this step doesn't consume any processing resource), but before doing that we have to check how many ingoing Transitions we have into the End step (see line 272). If the End step has only 1 ingoing Transition then the needed LQN Precedence has been already instantiated by the previous Step and the End step must not do anything; if instead the number of ingoing Transition is greater than one then we have to model a KLAPER merge of branches (coming from a Branch step) that is transformed into an LQN OR-Join precedence with ActivityList as pre-element (see line 275) and a single activity (the one transformed by the End step) as post-element (see line 276).

Another KLAPER base Step is the Wait step whose aim is to wait for some particular event; its rule is

```
290 /**
291  * transforms a Wait step
292  */
293 private transformStep(klaper::core::Wait wait, List[ActivityDef] activities, List[
        Precedence] precedences, lqn::Entry entry):
294         if(!(activities.exists(a|a.name == wait.name))) then // the step is
                transformed only once
295         (
296                 let activity = new lqn::ActivityDef:
297                 if(wait.in.size > 1) then // for (incoming) OR-join precedence
298                 (
299                         let in_precedence = new lqn::Precedence:
300                         in_precedence.setPreOR(wait.createActivityListIn()) ->
301                         in_precedence.setPost(wait.in.first().to.
                                createSingleActivityList()) ->
302                         precedences.add(in_precedence)
```

```
303                 ) ->
304                 activity.setName(wait.name) ->
305                 activity.setHostDemandMean(0.0) ->
306                 activity.setHostDemandCvsq(1.0) ->
307                 activity.setThinkTime(0.0) ->
308                 activity.setMaxServiceTime(0.0) ->
309                 activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
310                 activities.add(activity) ->
311                 if(wait.out.first().to.in.size == 1) then
312                 (
313                         let out_precedence = new lqn::Precedence:
314                         out_precedence.setPre(wait.createSingleActivityList()) ->
315                         out_precedence.setPost(wait.out.first().to.
                                createSingleActivityList()) ->
316                         precedences.add(out_precedence)
317                 ) ->
318                 wait.out.first().to.transformStep(activities, precedences, null)
319         );
```

as already seen with the End step, the first thing to do (see line 294) is to check if the step has already been transformed. After that from line 297 to line 303 we set the possible OR-Join condition (see the End step for an explanation). Then from line 304 to line 310 we set all the ActivityDef attributes (again the *hostDemandMean* attribute is set to 0 because this is only a control step that doesn't require any processing resource). To complete the transformation from line 311 to line 317 we set the standard simple Precedence in the case the next step has only one incoming Transition, otherwise it will be responsibility of the next Step to allocate the right Precedence.

The KLAPER Branch step transformation rule is not so different from those seen until now

```
321 /**
322  * transforms a Branch step
323  */
324 private transformStep(klaper::core::Branch branch, List[ActivityDef] activities,
        List[Precedence] precedences, lqn::Entry entry):
325         if(!(activities.exists(a|a.name == branch.name))) then // the step is
                transformed only once
326         (
327                 let activity = new lqn::ActivityDef:
328                 let out_precedence = new lqn::Precedence:
329                 if(branch.in.size > 1) then // for (incoming) OR-join precedence
330                 (
331                         let in_precedence = new lqn::Precedence:
332                         in_precedence.setPreOR(branch.createActivityListIn()) ->
333                         in_precedence.setPost(branch.in.first().to.
                                createSingleActivityList()) ->
```

```
334                         precedences.add(in_precedence)
335                 ) ->
336                 activity.setName(branch.name) ->
337                 activity.setHostDemandMean(0.0) ->
338                 activity.setHostDemandCvsq(1.0) ->
339                 activity.setThinkTime(0.0) ->
340                 activity.setMaxServiceTime(0.0) ->
341                 activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
342                 activities.add(activity) ->
343                 out_precedence.setPre(branch.createSingleActivityList()) ->
344                 out_precedence.setPostOR(branch.createOrList()) ->
345                 precedences.add(out_precedence) ->
346                 branch.out.to.transformStep(activities, precedences, null)
347         );
```

at line 325 we check if the Step has been already transformed. Then from line 329 to line 335 we check the OR-Join condition and from line 336 to line 342 we set all ActivityDef attributes (one more time see the *hostDemandMean* attribute set to 0). The differences from the other transformation rule seen so far is that this time a new kind of precedence is created: the pre-element (see line 343) is a single activity list (already used for other Steps), but the post-element (see line 344) is an ORList element that expresses each outgoing transition with a particular type of LQN activity called ActivityOR characterized by a *prob* attribute that defines the probability of a specific branch.

The transformation rule used for the KLAPER Fork step is very similar to that just seen for the Branch step (and this is easy to understand if we consider that the two Steps are very similar, the only difference is that the Branch step describes alternative flows while the Fork step describes parallel flows); the transformation rule is

```
550 /**
551  * transforms a Fork step
552  */
553 private transformStep(klaper::core::Fork fork, List[ActivityDef] activities, List[
        Precedence] precedences, lqn::Entry entry):
554         if(!(activities.exists(e|e.name == fork.name))) then // the step is
                transformed only once
555         (
556                 let activity = new lqn::ActivityDef:
557                 let out_precedence = new lqn::Precedence:
558                 if(fork.in.size > 1) then // for (incoming) OR-join precedence
559                 (
560                         let in_precedence = new lqn::Precedence:
561                         in_precedence.setPreOR(fork.createActivityListIn()) ->
```

```
562                         in_precedence.setPost(fork.in.first().to.
                                createSingleActivityList()) ->
563                         precedences.add(in_precedence)
564                 ) ->
565                 activity.setName(fork.name) ->
566                 activity.setHostDemandMean(0.0) ->
567                 activity.setHostDemandCvsq(1.0) ->
568                 activity.setThinkTime(0.0) ->
569                 activity.setMaxServiceTime(0.0) ->
570                 activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
571                 activities.add(activity) ->
572                 out_precedence.setPre(fork.createSingleActivityList()) ->
573                 out_precedence.setPostAND(fork.createActivityListOut()) ->
574                 precedences.add(out_precedence) ->
575                 fork.out.to.transformStep(activities, precedences, null)
576         );
```

where we have all the elements already seen: the check if the Step has been already transformed (line 554), the OR-Join condition for a number of incoming Transitions greater than 1(from line 558 to line 564), the ActivityDef attributes configuration (from line 565 to line 571) and finally the outgoing precedence, but here we have a specific behavior for the Fork step.

For each Fork step we need a corresponding Join step. The transformation rule for the KLAPER Join step is

```
579 /**
580  * transforms a Join step
581  */
582 private transformStep(klaper::core::Join join, List[ActivityDef] activities, List[
        Precedence] precedences, lqn::Entry entry):
583         if(!(activities.exists(e|e.name == join.name))) then // the step is
                transformed only once
584         (
585                 let activity = new lqn::ActivityDef:
586                 let in_precedence = new lqn::Precedence:
587                 in_precedence.setPreAND(join.createAndJoinList()) ->
588                 in_precedence.setPost(join.createSingleActivityList()) ->
589                 precedences.add(in_precedence) ->
590                 activity.setName(join.name) ->
591                 activity.setHostDemandMean(0.0) ->
592                 activity.setHostDemandCvsq(1.0) ->
593                 activity.setThinkTime(0.0) ->
594                 activity.setMaxServiceTime(0.0) ->
595                 activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
596                 activities.add(activity) ->
597                 if(join.out.first().to.in.size == 1) then
598                 (
599                         let out_precedence = new lqn::Precedence:
600                         out_precedence.setPre(join.createSingleActivityList()) ->
```

```
601                     out_precedence.setPost(join.out.first().to.
                            createSingleActivityList()) ->
602                     precedences.add(out_precedence)
603                 ) ->
604                 join.out.first().to.transformStep(activities, precedences, null)
605         );
```

where at line 583 we have the usual if condition to check if the step has been already transformed to avoid loops. Then this time we have no condition on the number of incoming Transitions because the role of the Join step is exactly that of merging parallel execution flows; to do this we need a new type of precedence composed by an LQN AndJoinList (that is a list of incoming activities with the specification of the number of completed activities needed before going to the next activity) as pre-element (see line 587) and a simple single LQN activity as post-element (see line 588). Once this Precedence is properly instantiated, as usual we can set all ActivityDef attributes (*hostDemandMean* always set to 0, see lines from 590 to 596) and instantiate the final simple Precedence if the next step has only one incoming Transition (see lines from 597 to 603).

Until now we have seen all the transformation rules used for KLAPER control steps. Since they are all control steps, they are all characterized by a value for the *hostDemandMean* attribute equal to 0; this is done because control steps are only used to manage the execution flow of KLAPER activities and therefore don't waste any processing resource, that is they do not consume time. But a performance evaluation model need a way to express the time consumed by actions accomplished by the system; in KLAPER this idea is expressed using Activities and ServiciControl and now we will see how these meta classes can be transformed into LQN concepts.

KLAPER Activities are used to represent three different concepts: simple activities, loops and nested behaviors. The transformation rule used to map a KLAPER Activity into some LQN concepts directly reflects this multiple meaning

```
349 /**
350  * transforms an Activity step
351  */
352 private transformStep(klaper::core::Activity activity, List[ActivityDef] activities
    , List[Precedence] precedences, lqn::Entry entry):
353         if(!(activities.exists(e|e.name == activity.name))) then // the step is
                transformed only once
354         (
355                 let act = new lqn::ActivityDef:
356                 let loop = new lqn::ActivityDef:
357                 let loop_name = activity.name + "_loop":
```

```
358                 let is_loop = (activity.repetitions == null) ? (false) : (!(
                        activity.repetitions.getMean() == 1.0)):
359
360                 if(activity.in.size > 1) then // for (incoming) OR-join precedence
361                 (
362                         let in_precedence = new lqn::Precedence:
363                         in_precedence.setPreOR(activity.createActivityListIn()) ->
364                         // for a single activity
365                         in_precedence.setPost(activity.in.first().to.
                                createSingleActivityList()) ->
366                         precedences.add(in_precedence)
367                 ) ->
368
369                 if(is_loop || (activity.nestedBehavior != null)) then // loop
                        condition
370                 (
371                         let loop = new lqn::ActivityDef:
372                         let loop_precedence = new lqn::Precedence:
373                         let loop_list = new lqn::ActivityLoopList:
374
375                         loop.setName(activity.name) ->
376                         loop.setHostDemandMean(0.0) ->
377                         loop.setHostDemandCvsq(1.0) ->
378                         loop.setThinkTime(0.0) ->
379                         loop.setMaxServiceTime(0.0) ->
380                         loop.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
381                         activities.add(loop) ->
382
383                         loop_precedence.setPre((new lqn::SingleActivityList).
                                setActivity((new lqn::Activity).setName(activity.name))
                                ) ->
384                         loop_list.setEnd(activity.out.first().to.name) ->
385                         if(activity.nestedBehavior == null) then // simple activity
386                         (
387                                 let loop_single_activity = new lqn::ActivityDef:
388                                 let activity_loop = new lqn::ActivityLoop:
389
390                                 loop_single_activity.setName(activity.name + "
                                        _loop_item") ->
391                                 loop_single_activity.setHostDemandMean(activity.
                                        internalExecTime.getMean()) ->
392                                 loop_single_activity.setHostDemandCvsq(1.0) ->
393                                 loop_single_activity.setThinkTime(0.0) ->
394                                 loop_single_activity.setMaxServiceTime(0.0) ->
395                                 loop_single_activity.setCallOrder(lqn::CallOrder::
                                        STOCHASTIC) ->
396                                 activities.add(loop_single_activity) ->
397                                 activity_loop.setName(activity.name + "_loop_item")
                                         ->
398                                 activity_loop.setCount(activity.repetitions.getMean
                                        ()) ->
```

```
399                         loop_list.activity.add(activity_loop)
400                     ) ->
401 
402                     if(activity.nestedBehavior != null) then // nested behavior
403                     (
404                         let activity_loop = new lqn::ActivityLoop:
405 
406                         buildActivitiesGraph(activity.nestedBehavior,
                                activities, precedences, null) ->
407                         activity_loop.setName(activity.nestedBehavior.step.
                                typeSelect(klaper::core::Start).first().name)
                                ->
408                         (activity.repetitions != null) ? activity_loop.
                                setCount(activity.repetitions.getMean()) :
                                activity_loop.setCount(1.0) ->
409                         loop_list.activity.add(activity_loop)
410                     ) ->
411                     loop_precedence.setPostLOOP(loop_list) ->
412                     precedences.add(loop_precedence)
413                 )
414                 else // single activity
415                 (
416                     if(activity.nestedBehavior == null) then // simple activity
417                     (
418                         act.setName(activity.name) ->
419                         act.setHostDemandMean(activity.internalExecTime.
                                getMean()) ->
420                         act.setHostDemandCvsq(1.0) ->
421                         act.setThinkTime(0.0) ->
422                         act.setMaxServiceTime(0.0) ->
423                         act.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
424                         activities.add(act)
425                     )
426                 ) ->
427 
428                 // for a single activity
429                 if(activity.out.first().to.in.size == 1) then
430                 (
431                     let out_precedence = new lqn::Precedence:
432                     out_precedence.setPre(activity.createSingleActivityList())
                            ->
433                     out_precedence.setPost(activity.out.first().to.
                            createSingleActivityList()) ->
434                     precedences.add(out_precedence)
435                 ) ->
436 
437                 activity.out.first().to.transformStep(activities, precedences, null
                        )
438         );
```

we start with a preparatory phase where at line 353 we find the usual if condition

used to check if a Step has already been transformed needed to avoid loops, at line 358 we check if we are transforming a loop (a mean number of repetitions greater than 1.0 represents a loop condition) and finally from line 260 to line 367 we manage the usual OR-Join condition. From line 369 to line 413 we handle loop and nested behavior conditions. More precisely from line 371 to line 381 we instantiate a new LQN ActivityDef element that is only a dummy activity used to delimit the loop or the nested behavior (the name use for the activity is related to loops, but it is only a name and this strategy can work both for loops and for nested behaviors); once the dummy loop activity has been created we have to link it to other activities of the same Task and to do this we create a new Precedence specific for loops characterized by a simple activity as pre-element (see line 383), a list of parallel activities to execute for the loop (see line 411) and a simple activity as the activity to execute at the end of the loop (see line 411). When the specific loop Precedence has been instantiated we have a distinction into the way to transform simple loops and the way to transform nested behaviors. Simple loops (considered as loops composed by a single activity) are transformed from line 385 to line 400 where we create a new ActivityDef instance and we set all its attributes, including the *hostDemandMean* attribute that this time is set to a value computed from the *internalExecTume* of the source KLAPER step because KL; we have to note that before completing the section of simple loops from line 397 to line 399 we have to add a reference into the list of activities done by the current loop and expressed by the loop Precedence of lines 383, 384, 411 and 412 and at line 398 we have to set the mean number of times the loop is iterated. If we haven't a loop a single simple activity but we have to transform a nested behavior we can simply recursively call the buildActivitiesGraph() function (line 406) we already discussed for KLAPER Behaviors; when the nested behavior is completely transformed we have to set the mean number of iterations of the loop (see line 408) and link all the created elements to the loop Precedence as already seen for simple loops. One thing is very important to note, LQN loops can specify a number of activities that can be iterated at the same time; KLAPER doesn't have this feature, it is only capable of expressing loops composed by one step or by a graph of steps, but these steps are executed in sequence and not in parallel at least that you don't use a Fork step into the graph; therefore into KLAPER to LQN transformations we use LQN loops not at their full power, but it is only because we don't need that. Now come back to the KLAPER Activity

step transformation with the simplest case, that of a simple single activity; this time the transformation is very simple (see lines from 415 to 426) because we only need to set the ActivityDef attributes taking care to set the *hostDemandMean* attribute to the mean value of the *internalExecutionTime* of the related KLAPER Activity step. Before closing this transformation rule from line 429 to line 435 we instantiate the final single Precedence needed if the next Step has only one incoming Transition as already done for other KLAPER Steps transformation rules.

As discussed in section 9.1 the KLAPER Activity meta class has a very important extension named ServiceControl that is used for service calls and for sendig signals; its transformation rule is

```
441 /**
442  * transforms a ServiceControl step
443  */
444 private transformStep(klaper::core::ServiceControl service, List[ActivityDef]
        activities, List[Precedence] precedences, lqn::Entry entry):
445         if(!(activities.exists(e|e.name == service.name))) then // the step is
                transformed only once
446         (
447                 let act = new lqn::ActivityDef:
448                 let loop = new lqn::ActivityDef:
449                 let loop_name = service.name + "_loop":
450                 let is_loop = (service.repetitions == null) ? (false) : (!(service.
                    repetitions.getMean() == 1.0)):
451 
452                 if(service.in.size > 1) then // for (incoming) OR-join precedence
453                 (
454                         let in_precedence = new lqn::Precedence:
455                         in_precedence.setPreOR(service.createActivityListIn()) ->
456                         // for a single activity
457                         in_precedence.setPost(service.in.first().to.
                            createSingleActivityList()) ->
458                         precedences.add(in_precedence)
459                 ) ->
460 
461                 if(is_loop || (service.nestedBehavior != null)) then // loop
                    condition
462                 (
463                         let loop = new lqn::ActivityDef:
464                         let loop_precedence = new lqn::Precedence:
465                         let loop_list = new lqn::ActivityLoopList:
466 
467                         loop.setName(service.name) ->
468                         loop.setHostDemandMean(0.0) ->
469                         loop.setHostDemandCvsq(1.0) ->
470                         loop.setThinkTime(0.0) ->
471                         loop.setMaxServiceTime(0.0) ->
```

```
472                         loop.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
473                         activities.add(loop) ->
474 
475                         loop_precedence.setPre((new lqn::SingleActivityList).
                                setActivity((new lqn::Activity).setName(service.name)))
                                 ->
476                         loop_list.setEnd(service.out.first().to.name) ->
477                         if(service.nestedBehavior == null) then // simple activity
478                         (
479                                 let loop_single_activity = new lqn::ActivityDef:
480                                 let activity_loop = new lqn::ActivityLoop:
481 
482                                 loop_single_activity.setName(service.name + "
                                        _loop_item") ->
483                                 loop_single_activity.setHostDemandMean(service.
                                        internalExecTime.getMean()) ->
484                                 loop_single_activity.setHostDemandCvsq(1.0) ->
485                                 loop_single_activity.setThinkTime(0.0) ->
486                                 loop_single_activity.setMaxServiceTime(0.0) ->
487                                 loop_single_activity.setCallOrder(lqn::CallOrder::
                                        STOCHASTIC) ->
488                                 activities.add(loop_single_activity) ->
489                                 activity_loop.setName(service.name + "_loop_item")
                                        ->
490                                 activity_loop.setCount(service.repetitions.getMean
                                        ()) ->
491                                 loop_list.activity.add(activity_loop)
492                         ) ->
493 
494                         if(service.nestedBehavior != null) then
495                         (
496                                 let activity_loop = new lqn::ActivityLoop:
497 
498                                 buildActivitiesGraph(service.nestedBehavior,
                                        activities, precedences, null) ->
499                                 activity_loop.setName(service.nestedBehavior.step.
                                        typeSelect(klaper::core::Start).first().name)
                                        ->
500                                 (service.repetitions != null) ? activity_loop.
                                        setCount(service.repetitions.getMean()) :
                                        activity_loop.setCount(1.0) ->
501                                 loop_list.activity.add(activity_loop)
502                         ) ->
503                         loop_precedence.setPostLOOP(loop_list) ->
504                         precedences.add(loop_precedence)
505                 )
506                 else // single activity
507                 (
508                         if(service.nestedBehavior == null) then // simple activity
509                         (
510                                 let call = new lqn::ActivityMakingCall:
```

```
511
512                                     // activity attributes
513                                     act.setName(service.name) ->
514                                     act.setHostDemandMean(service.internalExecTime.
                                            getMean()) ->
515                                     act.setHostDemandCvsq(1.0) ->
516                                     act.setThinkTime(0.0) ->
517                                     act.setMaxServiceTime(0.0) ->
518                                     act.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
519                                     // service control attributes
520                                     call.setFanin(1) ->
521                                     call.setFanout(1) ->
522                                     (service.binding.call != null) ?
523                                             call.setDest(service.binding.call.name) :
                                                    call.setDest(service.binding.signal.
                                                    name) ->
524                                     call.setCallsMean(1.0) ->
525                                     (service.isSynch) ?
526                                     (
527                                             act.synchCall.add(call)
528                                     )
529                                     :
530                                     (
531                                             act.asynchCall.add(call)
532                                     ) ->
533                                     activities.add(act)
534                             )
535                     ) ->
536
537                     // for a single activity
538                     if(service.out.first().to.in.size == 1) then
539                     (
540                             let out_precedence = new lqn::Precedence:
541                             out_precedence.setPre(service.createSingleActivityList())
                                    ->
542                             out_precedence.setPost(service.out.first().to.
                                    createSingleActivityList()) ->
543                             precedences.add(out_precedence)
544                     ) ->
545
546                     service.out.first().to.transformStep(activities, precedences, null)
547             );
```

this transformation rule is identical to that of the Activity meta class but with a difference in the case of the mapping of a single simple activity (in other words when we have no loops and no nested behaviors). Examining lines from 507 to 535 we can see that from line 513 to line 518 we simply set all the LQN ActivityDef meta class attributes, then we start setting all the necessary stuff required by a request of a service from another resource or by a signal raising. At lines 520 and

521 we set the *fanin* and the *fanout* of the LQN activity call; they are both set to 1 because into the KLAPER to LQN transformation we always map a one to one relationship between clients and servers, while LQN is capable with the multiplicity and the replication concepts to map also other kind of relationships. At lines 522 and 523 we set the name of the called LQN Entry (or Service if we use the KLAPER nomenclature) if we are modeling a service call, otherwise we set the name of the LQN activity modeling the KLAPER Wait step. Then from line 525 to line 532 we add the ActivityMakingCall instance that represents the call to the right list depending on the synchronous or asynchronous nature of the call. We skipped line 524 because the trasnformation is not yet able to set the mean time an LQN call is done; this information should be computed considering the KLAPER actual and formal parameters concepts, but actually we have to complete KLAPER parameters transformation; therefore we simply set the mean number of a call to 1 but this value has to be changed by hand after the final lqns input file is produced.

With the ServiceControl step we have transformed all the KLAPER meta model elements, but we still have to do the second phase of the transformation: the processors reconfiguration. Until now we have allocated all the LQN Tasks into a single dummy processor (except for workloads that have their own processors), but we have to solve this situation creating the rigth Processors and allocating each Tesk on its own Processor. To do this first of all we create the needed Processor with the following transformation rule

```
669 /**
670  * creates the real processors and reconfigure tasks from the dummy processor
671  */
672 private lqn::LqnModel reconfigureProcessors(lqn::LqnModel temp_model, klaper::core
        ::KlaperModel m):
673 //      //---- right now hardware resources names are hardcoded but they should be
        defined by the user!!! TODO
674         temp_model.processor.addAll(m.resource.select(e|{'cpu', 'network', 'disk'}.
                contains(e.type)).transformProcessor()) ->
675 
676         // removes the dummy processor
677         temp_model.setProcessor(temp_model.processor.withoutFirst()) ->
678         temp_model;
```

where at line 674 we select all the KLAPER resources named “cpu”, “network” or “disk” (that is all are hardware resources), these resources are the transformed into LQN Processors using the transformProcessor() function; currently the name of hardware resources has to match one of the three hardcoded possibilities (“cpu”,

“network” and “disk”) but the selection should be up to the tool user, may be using a form to specify the names of hardware resources for the provided input model; the problem is that KLAPER doesn’t make any difference between hardware and software resources but LQN does and therefore we have two possibilities: to hardcode the name of the hardware resources (this is how thing work now, but the modeler has to respect the chosen convention) or to leave to the modeler the capacity to specify which are the hardware resources (but this approach requires a support from the tool user during the transformation). After creating the right Processors at line 677 we can remove from the model the dummy Processor using the function withoutFirst() that applied to a list returns the same list but without the first element.

After all the needed Processors have been created we can relocate Tasks according to the real deployment of the system using the transformation rule

```
681  /**
682   * creates an lqn processor from an hardware klaper resource
683   */
684  private create lqn::Processor this transformProcessor(klaper::core::Resource r):
685          this.setName(r.name) ->
686          this.setMultiplicity(1) ->
687  //      this.setSpeedFactor(0.0) ->
688          this.setReplication(1) ->
689  //      this.setQuantum(0.1) ->
690          (r.schedulingPolicy != null) ? this.setScheduling(r.schedulingPolicy.
             transformScheduling()) : null ->
691  //      this.setScheduling(lqn::SchedulingType::ps) ->
692          this.setQuantum(0.1 * 0.1 * 0.1 * 0.1 * 0.1) -> // only because 0.00001
             give a parse error! but why???
693
694          // adds the main task running on this processor (the task that represents
             hardware services)
695          this.task.add(r.transformTask()) -> // it uses caching!!!!
696
697          // looks for all the tasks that use the previous one
698          this.task.addAll(
699                  ((klaper::core::KlaperModel)(r.eContainer)).resource.select(e | e
                     != r).select(e | e.offeredService.behavior.step.typeSelect(
                     klaper::core::ServiceControl).binding.call.name.intersect(r.
                     transformTask().entry.name).size > 0).transformTask()
700          ) ->
701          this;
```

where the truly interesting operations happen at the last two lines. At line 695 we add to the current Processor the Task that represents its effective processing power; to understand that we have to consider that in KLAPER each Resource representing an hardware resource has always at least a single Service that represents

the specific action done by that hardware element, for example cpus always have a "process" Service, networks always have a "transmit" Service and disks always have a "readWriteAccess" Service[2]. After that with the code of line 699 we select all the Tasks containing calls to the Task of line 695 and we put them into the current Processor because if they use the Task that represents the processing action of a Processor they have to be run on this Processor. To note that all the Tasks created at line 695 and line 699 are not truly instantiated, because the were previously created and thanks to the caching system of Xtend the underlying engine understand that a specific Task has been already created and simply returns a reference to this Task rather than creating a new one, so also performances of the final tool are guaranteed.

9.4 Using the LQN meta model

Now that we know how to transform a KLAPER model into an LQN model we can run our transformations as needed, but to make them useful we still need a way to transform from the LQN meta model to the input file used by lqns (the LQN analytical solver). The model to text transformation from LQN to the input format used by lqns is not discussed here simply because it is very simple considering that the LQN meta model has been built starting from the xsd schema used by lqns itself, therefore transforming from an LQN model into the xml file format used by lqns in most cases is a one to one transformation that simply adds some xml tags. However you can find more details about the M2T transformation for LQN inspecting the code at [47] into the svn browsing section.

When we have an xml input file representing the model to be solved, we can give this input to the lqns solver that modifies the input file adding the computed results for each entity (or better for each LQN Activity). The results are utilization and throughput and are computed for Activities, Tasks, Entries and Processors and are very useful to understand the behavior of the system and to find bottlenecks of the input model.

[2]This is not strictly dependent from the KLAPER meta model but rather it is a kind of convention to make things easily working

Chapter 10

A case study

In previous chapters we saw why and how KLAPER can be used to evaluate software quality indexes.

In this chapter we will see KLAPER in action. We will see, from the point of view of the final user, how we can practically use some tools developed around KLAPER to analyze and modify a software design of an example system.

10.1 Tool integration

KLAPER provides a number of tools that implement all the concepts seen in previous chapters. In particular KLAPER provides some Eclipse plugins that extend the functionalities of this powerful tool.

All the plugins are completely integrated into the Eclipse environment and the user doesn't need any knowledge of concepts like model driven, the KLAPER meta model or the technologies used to implement these plugins (openArchitectureWare for example) because everything is hidden behind a simple graphical user interface like the one shown in figure 10.1. All that is required is the knowledge needed to write a KLAPER model (but this is only needed because we haven't yet a working transformation from input models like UML to KLAPER, this is planned for the future) and the knowledge needed to understand the output of the different external tools (e.g. lqns or SHARPE) used for software quality analysis.

Available KLAPER tools are:

- KLAPER dsl Project Wizard: it is a complete wizard that helps the user in

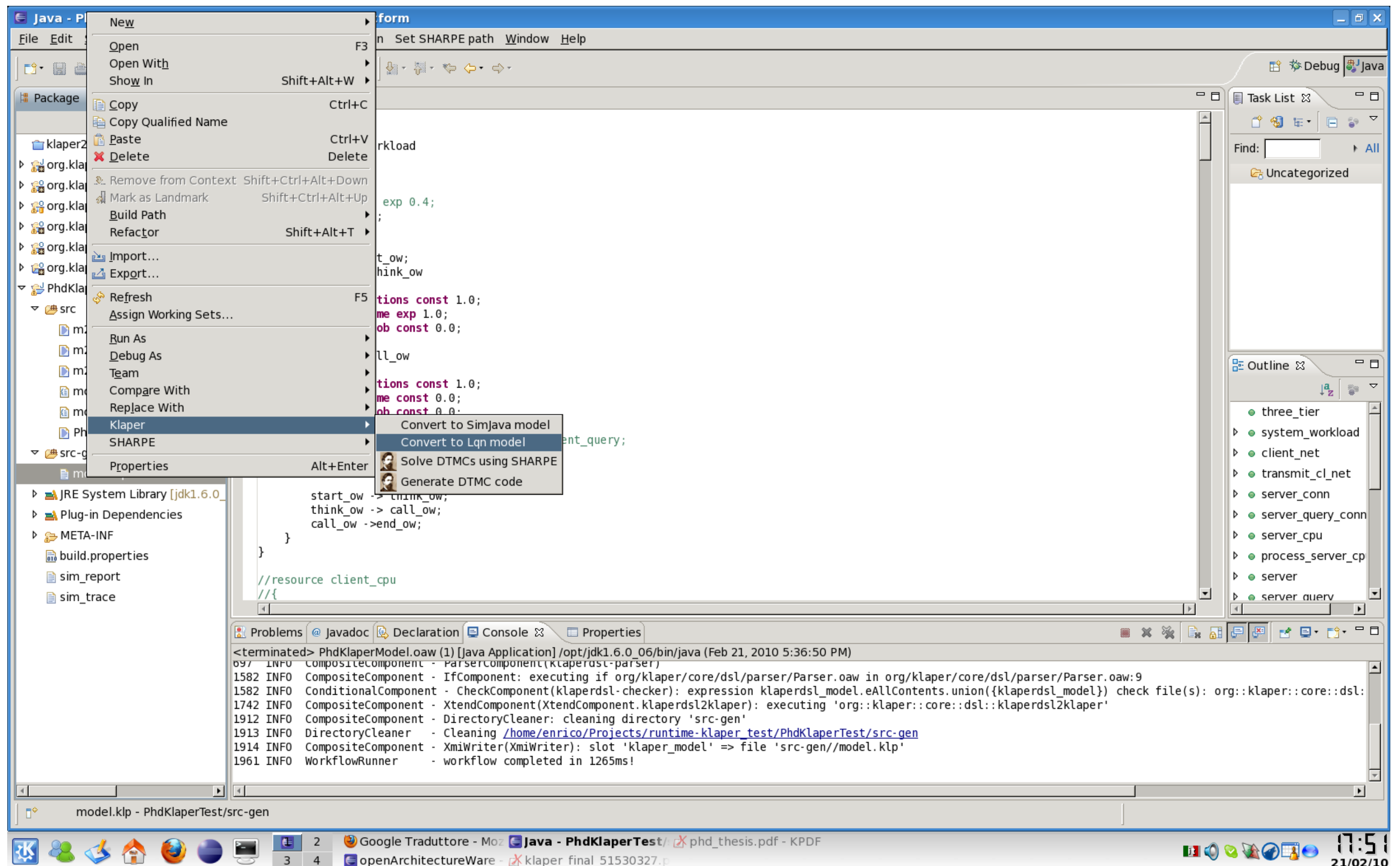

Figure 10.1. Integration of KLAPER into Eclipse and the related user interface

the creation of a KLAPER Project. It resolves all the plugin dependencies and create the initial files needed to successfully analyze the input model.

- KLAPER dsl Editor: it is a text editor for KLAPER models complete of syntax highlight and error checking at runtime (that means possible errors are detected and displayed while you are writing). See figure 10.2 for a sample screenshot.

- KLAPER to LQN transformation tool: you can launch a transformation from KLAPER to LQN simply right clicking on a KLAPER model (a file with a *.klp* extension), then from the context menu you have a practical KLAPER sub menu where you can choose the LQN transformation. As a result of this

simple action you obtain the LQN representation of your model ready to be used by the *lqns* solver.

- KLAPER to SimJava transformation tool: right clicking on a KLAPER model file you can choose the transformation to SimJava. As a result of this operation you obtain a java package containing all the source files needed to build the simulator; the only thing you have to do is to build the application and run the Subsystem java class (it can be directly run from inside Eclipse itself).

- KLAPER to DTMC transformation tool: like the other transformation tools you can launch a transformation simply right clicking on a KLAPER model file and choosing “Generate DTMC Code” from the KLAPER sub-menu of the obtained context menu. In addition from the context menu you have also a SHARPE sub-menu that runs the SHARPE tool for you and post process the obtained output results to make them more human readable.

All these tools can be easily used without any knowledge of model driven approaches; they may be used with just a little knowledge of KLAPER, but nothing is required about model transformations and the knowledge of the target analysis methodologies (even if it would be preferable).

Now that we have the tools, we need a practical example to see them working.

10.2 Description

As an example to see the KLAPER tools at work we choose a very simple and basic scenario derived from one of the case studies presented in [49]. Considering the simplicity of the example there would be no need to use any analysis tool to understand which are the design mistakes of the input software system, but we chose such a scenario exactly because it is so simple that results cannot be doubted.

As test-bed scenario we consider a simple client/server system with many clients connecting to a single server to have some kind of service. This server is simply an application server that makes some computations on the data provided by a backend database where all the needed information are stored. That is, this is a typical three-tier client/server system.

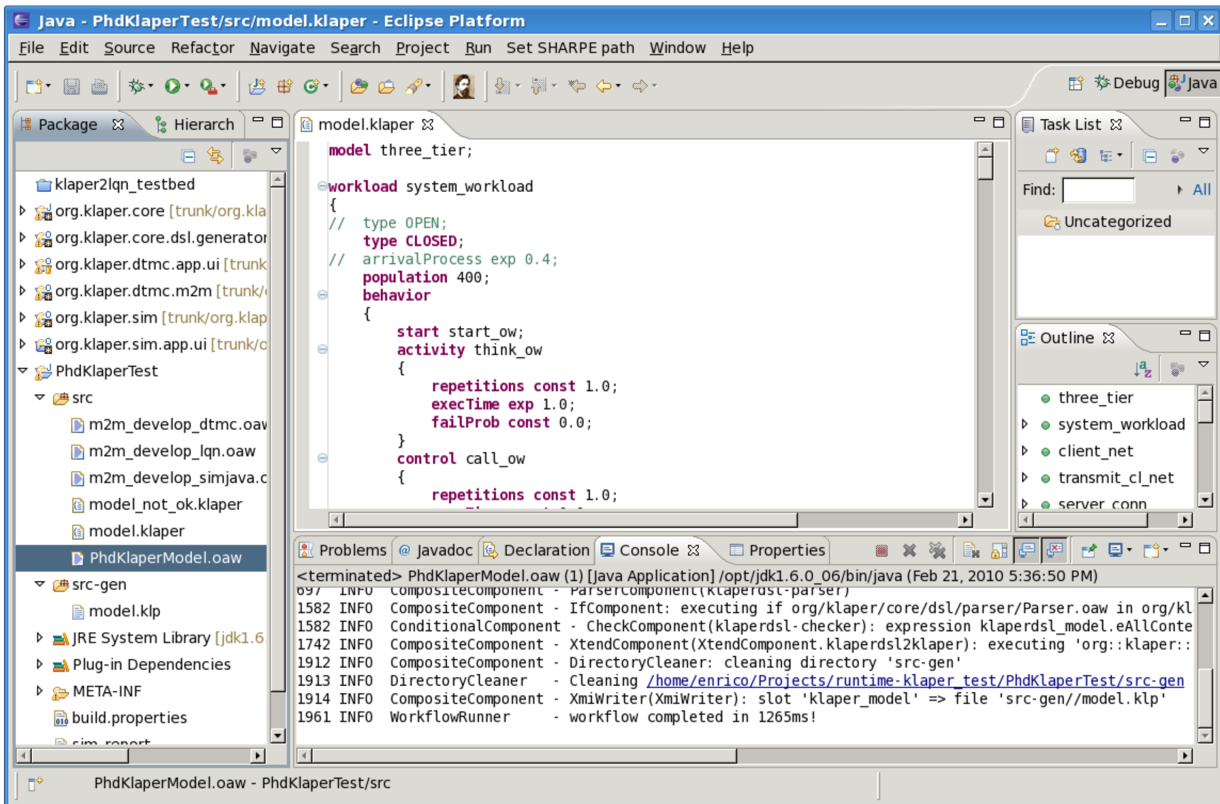

Figure 10.2. The KLAPER dsl Editor

We will analyze the overall system from the point of view of client applications; therefore our observation point will be the same of a web browser application, for example, and our first access point into the system will be the network that connects clients with the application server.

In our analysis we will use the following reliability and performance data as input to our computation:

- all cpus are pentium III with 500 Mhz (1354 MIPS) and a failure probability of 0.01%.
- client-server network is a 100 Mbps network with a failure probability of 0.7%.
- server-database network is a network based on a very old switch with 100 Mbps ports but a failure probability of 10%.

- the database disk is a typical IDE disk with a rate of 300 MBps and a failure probability of 2%.
- the system workload can be modeled by an open workload with an arrival rate of 0.87 requests per second.

and our software quality (non functional) requirements are:

- reliability of all the components (intended as the hardware and software building blocks) of the system must be greater than 90.0%.
- the service time (considered as the time needed to satisfy a request plus the time spent into the waiting queue of the resource providing the needed service) seen by clients must be less than 5 seconds.

10.2.1 The real input: UML

Even if actually KLAPER is not capable to handle an input different from its own domain specific language (presented in appendix A), we want to show what should be the real input of our analysis. For example we can express the system in terms of UML diagrams. To do this, for our specific example scenario we need at least two types of diagrams: deployment diagram and sequence diagram.

With the deployment diagram shown in figure 10.3 we can understand the deployment of the system that consists of a client node (actually we have more client nodes) on which the client application runs, a server node where the application server front-end runs and a database where the database system runs.

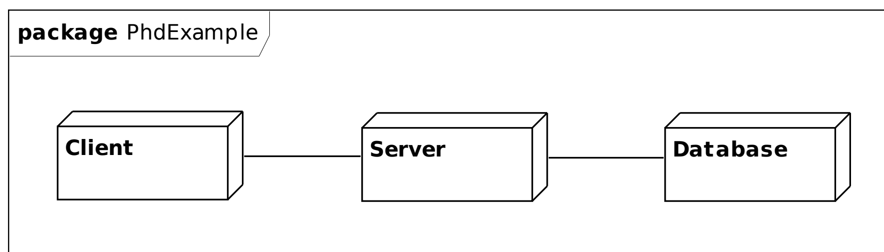

Figure 10.3. Example scenario UML deployment diagram

Simply looking at the deployment diagram shown in figure 10.3 we can understand that the three main components of our system are located on different nodes

and are linked using some kind of network; in this example we assume that we have two different networks: one to link clients and the server and another one to link the server and the database. This is a very useful information because networks must be modeled (using network connectors) into the KLAPER model to have a representation of the input system as close as possible to the real system.

Using only the deployment diagram is not enough, because we don't have any information about how services interact among them. To know this interaction we need some kind of behavioral description, like that provided by a sequence diagram.

In figure 10.4 we can see the sequence diagram of our input system; here we

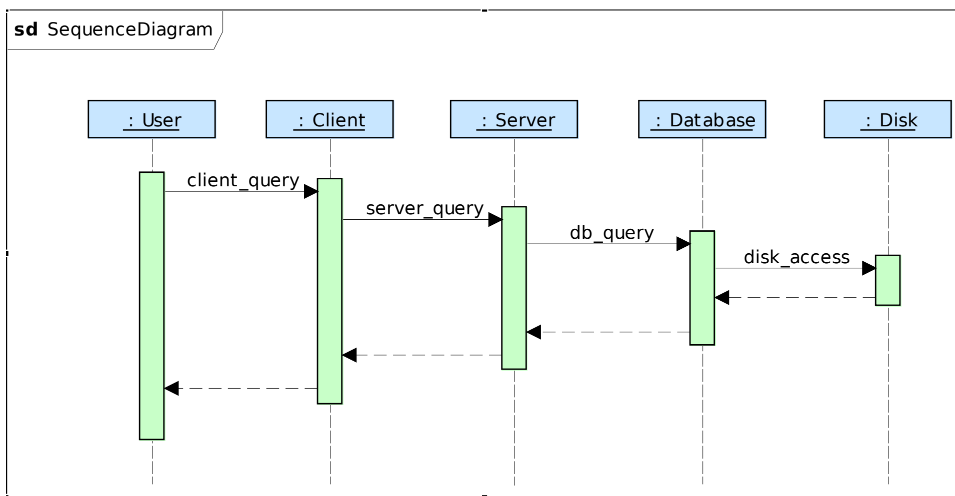

Figure 10.4. Example scenario UML sequence diagram

can see how each resource (the UML class instances) provides a service (the UML class methods) and how services require services provided by other resources. But this is not the only information we can extrapolate from the sequence diagram; an interesting thing to see in figure 10.4 is that database model should consist of a cpu, where the database application runs, and a disk (the sequence diagram identifies a specific class for the disk resource), where the database application has to access to fetch the information needed to its computations. Looking at the sequence diagram we can understand one more thing: in our input model all service calls are synchronous because all the arrows are full and this is the way UML uses to represent synchronous operation calls.

But the simple standard UML is not enough to represent all the information we need to build a KLAPER model, we need also performance and reliability data and UML is not capable of expressing such information. However one of the main peculiarities of UML is its capability to be extended; so we can use one of the UML extensions called *profiles* (the most used for this purpose are UML-SPT, see [43], and MARTE, see [42]) to augment the base UML representation with all the performance and reliability requirements and data we need.

10.3 The KLAPER representation

The UML model provided in section 10.2.1 should be the only input needed by KLAPER tools, but right now the transformation from UML to KLAPER is not ready yet and so we have to make it by hand. To do this task we are helped by the KLAPER dsl Editor, so we can express all the input system model into a textual format that can be automatically converted into a KLAPER model with a simple click of the mouse.

Even if not yet formalized, the mapping from UML to KLAPER should be based on the following general rules:

- Each UML component is mapped into a KLAPER Resource whose Services are directly converted from the provided interfaces. Behaviors of these Services are mapped from UML component state machine or activity diagram.

- UML diagrams modeling component use cases are used to create KLAPER Workloads whose Behaviors are derived from UML diagrams (for example sequence diagrams) modeling the use case dynamics.

- UML behavioral diagrams (state machine diagrams, sequence diagrams, collaboration diagrams and activity diagrams) are used to map KLAPER Services Behaviors. Each UML flow control element (like if conditions, loops and so on) present in these diagrams can be mapped to its related KLAPER control Step.

- Each access to a UML interface, but in general each use of a UML operation of a class, can be mapped to a KLAPER ServiceControl step.

- KLAPER meta classes attributes are manly derived from additional information applied to UML elements using some specific profile like UML-SPT or MARTE.

But applying the rules presented above is not sufficient to have a complete KLAPER mapping of the UML input model; there are two additional rules that we have to follow.

The first additional rule is that from UML nodes[1] and from UML nodes relationships we must create a special Resource with a Service that represents the basic service offered by the related hardware resource. In our model we have three of these services: transmit_cl_net (but also transmit_srv_net) that represents the sending of some bytes over a network, process_server_cpu (but also process_db_cpu) that represents the elaboration of a cpu, disk_access that represents an input/output operation towards a disk. All these services have not any explicit counterpart into the input UML model, they have been introduced in KLAPER to represent the way KLAPER describes an hardware resource (remember that KLAPER Resources can map indifferently an hardware or a software resource of the input system). More specifically so far we have three different types of special Resources in KLAPER:

- cpus: they represent a physical cpu and always present a single Service called *process* that represents the processing activity of the cpu. UML nodes that model some kind of computer usually generate a cpu Resource.

- networks: they represent a physical link between two nodes of the system that can be an ethernet link, but also a wi-fi connection, a serial cable or a communication bus and so on; they always present a single Service called *transmit* that represents a data transfer. Each UML communication path generate a network Resource (here "link" may be a better name than "network").

- disks: they represent a physical disk and always present a single Service called *disk_access* that represents input and output operations over the disk. UML nodes that model some kind of computer or storage device usually generate a disk Resource.

Example templates for cpus and networks special resources can be found in figures 10.5 and 10.6.

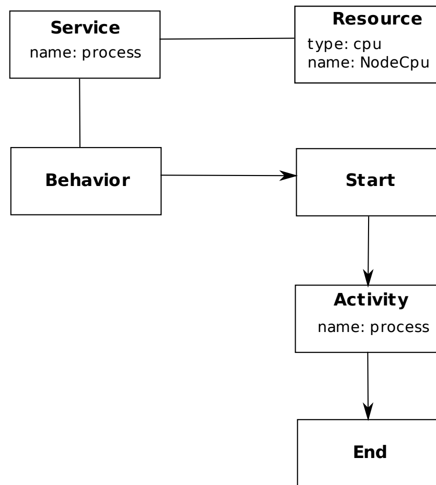

Figure 10.5. Template structure for KLAPER representation of a cpu special Resource

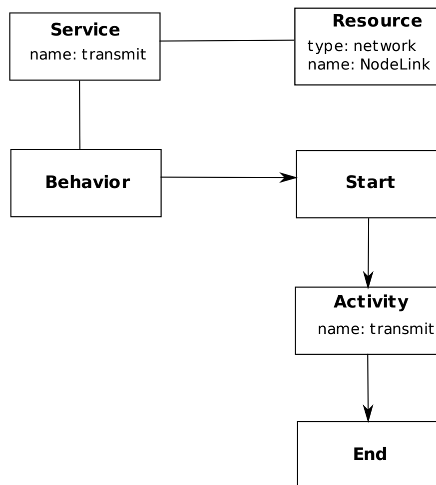

Figure 10.6. Template structure for KLAPER representation of a network special Resource

The second additional rule is about KLAPER connectors. Connectors are used to represent the data exchange actions (data sending and data receiving) over a link between two distinct entities connected using a specific physical means. Connectors are not a replacement for network special resources presented above, but they are a

[1]In UML deployment diagrams nodes represent hardware resources

level of abstraction over these; they are Services whose behavior is always composed in this way: a Start step, a ServiceControl Step that represents the sending of a request over a network special Resource, a ServiceControl Step that represents the call of the remote Service, a ServiceControl Step that represents the receiving of a response over the network special Resource and finally an End Step. Obviously this is the configuration used for synchronous calls, for asynchronous calls we don't have the last response ServiceCall. In figure 10.7 we can find a typical template example of KLAPER connectors.

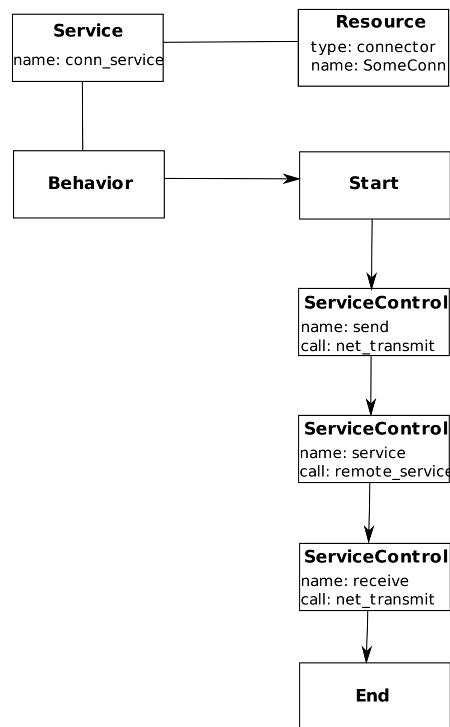

Figure 10.7. Template structure for a KLAPER synchronous connector

The complete listing for the input system provided in section 10.2.1, converted into a KLAPER model following the general rules just discussed and expressed using the KLAPER dsl language can be found in appendix B. In this source file and in the next sections we use some names to refer to services and hardware resources whose meaning is described in table 10.1.

Item	Description
client_net	The network that connects clients with the server (see figure 10.3)
server_cpu	The cpu of the server workstation (see figure 10.3).
server_net	The network that connects the server workstation with the database host (see figure 10.3).
db_cpu	The cpu of the database host (see figure 10.3).
disk	The physical disk of the database host (see figure 10.3).
server_query_connector	The connector service added to model the connection between clients and the server (see figure 10.7).
transmit_cl_net	The base service offered by the client_net hardware resource.
process_server_cpu	The base service offered by the server_cpu hardware resource.
server_query	The main service offered by the system server (see figure 10.4).
transmit_srv_net	The base service offered by the server_net hardware resource.
db_query_connector	The connector service added to model the connection between the server and the database (see figure 10.7).
process_db_cpu	The base service offered by the db_cpu hardware resource.
db_query	The main service offered by the system database (see figure 10.4).
disk_io	This is only a convenience service added to simplify the KLAPER representation of the input model, but it hasn't any reference to any hardware or software entity.
disk_access	The base service offered by the disk hardware resource.

Table 10.1. Items descriptions

10.4 The first run

Now that we have our KLAPER model we can apply a software quality analysis to the input system using the KLAPER project tools; running the three quality

analysis model, we obtain some results that are the starting point of our analysis.

We start with the DTMC transformation. In table 10.2 we can see the results directly reported from the SHARPE tool.

Item	Reliability
server_query_connector	78.25%
transmit_cl_net	99.30%
process_server_cpu	99.99%
server_query	79.36%
transmit_srv_net	90.00%
db_query_connector	79.37%
process_db_cpu	99.99%
db_query	97.99%
disk_io	98.00%
disk_access	98.00%

Table 10.2. DTMC Reliability

To validate the obtained results we can run the transformation to the SimJava based simulator that is also capable to evaluate the reliability of the system. The result of this new transformation is Java code that we can build and run to obtain an output file; from this output we can find the number of total events occurred for the server_query_connector service, that runs on the server_conn resource, and the related number of failures. Applying the equation:

$$Reliability = 1 - \frac{failures\ events\ number}{total\ events\ number} \tag{10.1}$$

we can compute the reliability for the server_query_connector service and therefore for the server_conn resource. The computed value is reported in table 10.3.

Item	Reliability
server_query_connector	78.58%

Table 10.3. SimJava Reliability

Looking at the results obtained so far, we can already make some considerations for reliability. Two different analysis methodologies (analytical solving for DTMC and simulation for SimJava) give a reliability for the server_query_connector around

78%, but we recall this is not sufficient to satisfy our reliability requirement that needs a value greater then 90.0%.

Now it is the turn of software performances computed using LQN and its analytical solver lqns. Running lqns on the transformed model we obtain the results reported in table 10.4 and 10.5.

Item	Service time (secs)	Throughput
server_query_connector	8.95028	0.869958
transmit_cl_net	0.00830947	1.73985
process_server_cpu	0.1	0.869895
server_query	8.92741	0.869926
transmit_srv_net	0.0168565	1.73974
db_query_connector	8.79875	0.869895
process_db_cpu	0.5	0.869857
db_query	8.72535	0.869872
disk_io	7.76877	0.869857
disk_access	1	0.869849

Table 10.4. LQN tasks service time and throughput

Item	Utilization
client_net	0.0142529
server_cpu	0.0869895
server_net	0.028504
db_cpu	0.434929
disk	0.869849

Table 10.5. LQN processors utilization

Looking at the LQN results we can see that the response time seen from the client point of view is quite high (it is greater than 8 seconds) and it is caused by the tasks of the disk resource (because all the tasks in the related call graph have such an high response time until those running on the disk resource), that incidentally has an utilization approximately of 87%.

10.5 Analyzing Reliability

To begin we decide to start with the reliability analysis. Examining the results reported in tables 10.2 and table 10.3 we can see the same values, so we are convinced of the goodness of these results; but DTMC results are more detailed than those of SimJava so we can concentrate only on table 10.2.

The server_query_connector service has a reliability value of 78.25%. We already said that this value is too low to satisfy the required reliability requirement so now we have to understand what causes such a result. Looking at table 10.2 we can see that only three services have a reliability below 90%: server_query_connector, server_query and db_query_connector. server_query_connector in its calls tree depends from server_query which in turn depends from db_query_connector; db_query_connector has a reliability of 79.37% and depends from db_query and transmit_srv_net: the former has a reliability of 97.99%, while the latter has a reliability of 90.0%. Apparently the best candidate to be the source of our problem is transmit_srv_net that has a lower reliability value, but how can a reliability of 90.0% of the transmit_srv_net service cause a final 79.37% value for the db_query_connector service? The answer is that db_query_connector calls transmit_srv_net twice: the first time for sending data for the request and the second time to receive data for the response of the synchronous call. So a simple 10.0% failure rate for the server-database network switch can cause a reliability value of 90% for this network and therefore a 79.37% as the service result that is propagated until clients.

To solve this situation we must lower the failure rate of the server-database network; we can achieve this replacing the old network switch with a better one that has a best reliability, that in this case means a lower failure rate. If for example we choose a network switch with a failure rate equal to 0.7% (before we had a failure rate of 10.0%), like the one used for the client-server network, running again the KLAPER tools and the analysis tools we obtain the results reported in table 10.6, 10.7, 10.8 and 10.9.

Analyzing the new results we can see that performance are completely unchanged and this is correct because we replaced our network switch with an equal one from the point of view of performance, but less faulty. If instead we analyze DTMC and SimJava results we can see that the results from the two analysis models are coherent and now all the reliabilities are above 90.0%; therefore the solution we tested can

Item	Reliability
server_query_connector	95.26%
transmit_cl_net	99.30%
process_server_cpu	99.99%
server_query	96.61%
transmit_srv_net	99.30%
db_query_connector	96.62%
process_db_cpu	99.99%
db_query	97.99%
disk_io	98.00%
disk_access	98.00%

Table 10.6. DTMC Reliability with the new network switch

Item	Reliability
server_query_connector	95.70%

Table 10.7. SimJava Reliability with the new network switch

be considered as a possible improvement of the initial system design that meets all system reliabilty requirements.

Item	Service time (secs)	Throughput
server_query_connector	8.95028	0.869958
transmit_cl_net	0.00830947	1.73985
process_server_cpu	0.1	0.869895
server_query	8.92741	0.869926
transmit_srv_net	0.0168565	1.73974
db_query_connector	8.79875	0.869895
process_db_cpu	0.5	0.869857
db_query	8.72535	0.869872
disk_io	7.76877	0.869857
disk_access	1	0.869849

Table 10.8. LQN tasks service time and throughput with the new network switch

Item	Utilization
client_net	0.0142529
server_cpu	0.0869895
server_net	0.028504
db_cpu	0.434929
disk	0.869849

Table 10.9. LQN processors utilization with the new network switch

10.6 Analyzing Performances

Now it is the turn of performance. But this time we don't start from tables 10.4 and 10.5 because they refers to our initial model that we already modified to solve reliability problems. Our analysis starting point will be the results reported in tables 10.8 and 10.9.

Before proceeding with the obtained results analysis we have to understand the meaning that these results take into the LQN world. The utilization defines the percentage of time a service is actually used (or the percentage of time it is active) related to the overall execution time; a low utilization means that a service is not very requested and/or that it is very fast to execute, while a high utilization means that a service has a lot of requests and/or it is quite slow to execute. Throughput defines the number of requests satisfied per second; it is a useful index if considered at the access point of the system or if considered in combination with some other index like utilization or service time. Service time defines the time a request waits into the service queue before being served plus the time needed to effectively satisfy the request.

In table 10.8 we find a service time of 8.95028 seconds for the service_query_connector service; it is quite high. If we look for the element that causes such a response time we can see that it is more or less present until the disk_io service. But the disk_io service simply depends from disk_access that has a service time of 1.0 second. How can a service that requires 1 second to execute generate 8.95028 of response time in the calling service? If we remember the definition of service time we gave for LQN few lines above, we can understand that many seconds are spent into the waiting queue of the guilty disk_io service. Our thesis is confirmed by the utilization reported into table 10.9 where the disk resource has an utilization equa to 86.98%

that is very high. We have found our bottleneck.

What is happening into the system is that a simple request from a client generates not so much work into the application server (we can see utilization that do not exceed 10%), but requires a lot of work into the database. Here every single request requires a fair amount of computation (the database cpu has an utilization of 43.49%) and a massive access to the disk (utilization value equal to 86.98%) to fetch the needed table rows for the queries.

To solve this situation we could change the disk with a more performing one (as we already done with the network switch to improve reliability), but we are already using a not so bad disk and buying a faster one would be too expensive for our project. This time we have to come back to the design phase and think our system in a way that it strongly reduces disk access. For example we could think about introducing a software caching system into our database to dramatically reduce the mean service time of the disk_access service form 1.0 second to 0.5 seconds. Running again all the transformations and the related tools we obtain the results reported in tables 10.10, 10.11, 10.12 and 10.13.

Item	Reliability
server_query_connector	95.26%
transmit_cl_net	99.30%
process_server_cpu	99.99%
server_query	96.61%
transmit_srv_net	99.30%
db_query_connector	96.62%
process_db_cpu	99.99%
db_query	97.99%
disk_io	98.00%
disk_access	98.00%

Table 10.10. DTMC Reliability with the new network switch and disk caching

Item	Reliability
server_query_connector	95.60%

Table 10.11. SimJava Reliability with the new network switch and disk caching

Item	Service time (secs)	Throughput
server_query_connector	1.92619	0.869991
transmit_cl_net	0.00830948	1.73997
process_server_cpu	0.1	0.869984
server_query	1.91207	0.869987
transmit_srv_net	0.0168566	1.73997
db_query_connector	1.80276	0.869984
process_db_cpu	0.5	0.869983
db_query	1.76949	0.869983
disk_io	0.885144	0.869983
disk_access	0.5	0.869982

Table 10.12. LQN tasks service time and throughput with the new network switch and disk caching

Item	Utilization
client_net	0.0142539
server_cpu	0.0869984
server_net	0.0285076
db_cpu	0.434991
disk	0.434991

Table 10.13. LQN processors utilization with the new network switch and disk caching

Analyzing the new obtained results we can see that DTMC reliability is completely unchanged (see table 10.10) and also SimJava reliability can be considered unchanged (see table 10.11), while for what concerns performance instead we have some differences.

Simply introducing a software caching system that reduces the disk access service time from 1 second to 0.5 seconds (per request), the overall service time seen from clients changes from almost 9 seconds to 1.92619 seconds. Obviously the overall system response time is strongly reduced because it is reduced the disk_access service time, but especially because this service time reduction in the disk services response is sufficient to dramatically reduce the time each request spends into the disk_io service waiting queue. This is also confirmed from the disk resource utilization that changes from an excessive 86.98% to a more reasonable 43.49%.

The throughput of all services is more or less unchanged due to the workload of

the system that in the time interval under analysis has never been such to cause some user (or better client) requests loss; all requests have been always served whitout the caching system, even if the response times where too long, and they do all the more so now that we have the caching system.

In this chapter we saw how we can use the tools developed around the KLAPER meta model jointly to some other external tools (SHARPE, SimJava and lqns) to evaluate performance and reliability of our system before producing a single line of code. From our analysis we saw how sometimes design problems can be solved acting at deployment level, for example choosing a node with some better characteristics, and how some other times problems can be solved simply returning to the design phase to change some elements of our software system. The analysis of possible actions that we can take to solve such situations are matters of the feedback analysis we briefly discussed in chapter 2; this is a field on which there are some research activities based on model driven approaches (see [64] and [46] for example), but they are out of the topic of this work. However we think that one fundamental element is clear: we can solve problems at the design phase strongly reducing the effects and the costs of wrong design decision or involuntary mistakes that we would be pay if the problems where discovered building and running the real system.

Chapter 11

Conclusions

11.1 What we have done

In this work we have seen that modern software systems are very hard to design and implement due to their complexity; discovering some design errors at a late software development cycle phase is a very bad event because the more late they are discovered the more they are hard to solve and require time and resources to be removed.

To address all these problems during the years some software quality analysis methodologies have been developed; they are able to identify potential problems without the need of the completed real system, simply starting from a (design) model of the system. But those methodologies are not very well known, especially from those engineers that are in charge of the software system design.

Model Driven approaches and technologies can help joining the design world and the analysis world building transformation rules between them. But there are different ways to apply these concepts.

KLAPER presents a possible solution about how to transform from design models to software quality analysis models introducing an intermediate meta model that can be used as a bridge from one world to the other reducing and simplifying transformations.

At the time this work is written KLAPER consists of a set of really working tools (Eclipse plugins) that provide:

- An implementation of the intermediate meta model.

- A domain specific language for the intermediate meta model with the related editor.
- A transformation engine to the DTMC models (and to the input format required by the SHARPE tool), for reliability evaluation.
- A transformation engine to the SimJava based simulator model (that produces Java code based on the SimJava library framework), for reliability and performance evaluation (performance evaluation with the SimJava simulator works for very small models, but it is still under development and need more validation).
- A transformation engine to the LQN models (and to the input format required by the lqns tool), for performance evaluation.

Even if these tools still have some bugs and limitation they are almost ready to be used in a real world environment, as we saw into the example scenario presented in chapter 10 where we discussed an example of a typical application of KLAPER tools.

11.2 What we will do

At present, the KLAPER tools are not yet bug free so in the near future we have to work in the direction of stability and usability because some particular model configurations cannot be solved yet.

Then we have to consider that some tools have to be completed. For example the LQN transformation engine right now is not yet able to handle KLAPER parameters and this could give some inconveniences because you have to build the KLAPER model in a not intuitive way to trick this problem.

Another aspect to consider is that the SimJava transformation engine produces some output models with a very limited usage range; the excessive number of Java threads used by the generated code forbids the use of this specific tool with large models or with models that have an high value for resources capacity.

Lately we have found another interesting route to follow. Indeed we realized that our approach to represent events in KLAPER is not enough versatile than we thought, so we are exploring different approaches to do that and this could lead to

some changes into the meta model, like for example the replacement of the Wait step with an approach based on Transitions activation and deactivation.

Finally it would be interesting to add some more target analysis methodologies to transform to. Because each analysis methodology has its own specific feature and helps solving a particular aspect of the input model, having as many methodologies as possible is a good idea.

11.3 Related works

During the years some similar works have been developed in the field of design phase software quality analysis evaluation; some of them are original works and some other are closely related to KLAPER itself. Here we present some of them:

- PUMA (see [48]): it is a project very similar to KLAPER in the sense that it takes as input a UML model and returns as output an LQN representation of it. The differences are that it doesn't use a true model driven approach and it doesn't use an intermediate meta model; moreover it has only LQN as its only output target analysis methodology.

- Palladio (see [34]): this is a project very similar to KLAPER whose aim is to provide a modeling language, with related tools, to design component-based software architectures. It uses the same approach based on model transformations, but without the intermediate meta model idea of KLAPER. It is more mature than KLAPER and has a very nice tool integration inside the Eclipse environment with a very usable man-machine interface.

- Q-impress (see [49]): is a project developed under the Seventh Framework Programme (see [1]) and its main aim is to create a method for quality-driven software development and evolution, where the consequences of design decisions and system resource changes on performance, reliability and maintainability can be foreseen through quality impact analysis and simulation. It uses the same approach of model transformations and to do that it is based on an intermediate meta model like in KLAPER. It is definitively a KLAPER in big; even KLAPER is one of the components of Q-impress, but its intermediate meta model is not yet so mature as the KLAPER one.

Appendix A

The Klaper dsl grammar

```
1  /**
2   *    file: klaperdsl.xtxt
3   *
4   *    author: Enrico Randazzo
5   *    organization: University of Rome "Tor Vergata"
6   *    email: enrico.randazzo@gmail.com
7   */
8
9  Model:
10     (types+=Type)*;
11
12 Type:
13     KlaperModel |
14     Resource |
15     Workload |
16     Service;
17
18
19 // KlaperModel class
20 KlaperModel:
21     "model" (name=ID)? ";" ;
22
23
24 // Resource class
25 Resource:
26     "resource" name=ID
27     "{"
28         (resource_attrs+=ResourceAttrs)*
29     "}";
30
31 ResourceAttrs:
32     Resource_Type |
33     Resource_Capacity |
34     Resource_SchedulingPolicy |
35     Resource_Description;
36
37 Resource_Type:
38     "type" type=STRING ";" ;
39
40 Resource_Capacity:
41     "capacity" capacity=DOUBLE ";" ;
42
43 Resource_SchedulingPolicy:
44     "scheduling" scheduling=SchedulingPolicyKind ";" ;
45
46 Enum SchedulingPolicyKind:
```

```
47        EarliestDeadlineFirst="EarliestDeadlineFirst" |
48        FIFO="FIFO" |
49        FixedPriority="FixedPriority" |
50        LeastLaxityFirst="LeastLaxityFirst" |
51        RoundRobin="RoundRobin" |
52        TimeTableDriven="TimeTableDriven";
53
54    Resource_Description:
55        "description" description=STRING ";" ;
56
57
58    // Workload class
59    Workload:
60        "workload" name=ID
61        "{"
62            (workload_attrs+=WorkloadAttrs)*
63        "}";
64
65    WorkloadAttrs:
66        Workload_Type |
67        Workload_ArrivalProcess |
68    //    Workload_ArrivalProcessParams |
69        Workload_Population |
70        Workload_InitialResource |
71        Behavior;
72
73    Workload_Type:
74        "type" type=WorkloadType ";" ;
75
76    Enum WorkloadType:
77        OPEN="OPEN" |
78        CLOSED="CLOSED";
79
80    Workload_Population:
81        "population" population=INT ";" ;
82
83    Workload_InitialResource:
84        "initialResource" initialResource=STRING ";" ;
85
86    Workload_ArrivalProcess:
87        "arrivalProcess" arrivalProcess=ProbabilityDistributionFunction ";" ;
88
89    ProbabilityDistributionFunction:
90        NormalDistributionFunction |
91        PoissonDistributionFunction |
92        UniformDistributionFunction |
93        ExpDistributionFunction |
94        ConstantDistributionFunction;
95
96    NormalDistributionFunction:
97        "normal" mean=MathExpr "," standDev=MathExpr;
98
99    PoissonDistributionFunction:
100       "poisson" mean=MathExpr;
101
102   UniformDistributionFunction:
103       "uniform" "[" min=MathExpr "," max=MathExpr "]";
104
105   ExpDistributionFunction:
106       "exp" mean=MathExpr;
107
108   ConstantDistributionFunction:
109       "const" value=MathExpr;
110
111
112   // Service class
113   Service:
114       "service" name=ID
115       "{"
116           (service_attrs+=ServiceAttrs)*
```

```
117     "}";
118
119 ServiceAttrs:
120     Service_SpeedAttr |
121     Service_FailAttr |
122     Service_Description |
123     Behavior |
124     Service_Resource |
125     FormalParams;
126
127 Service_SpeedAttr:
128     "speedAttr" speedAttr=DOUBLE ";" ;
129
130 Service_FailAttr:
131     "failAttr" failAttr=DOUBLE ";" ;
132
133 Service_Description:
134     "description" description=STRING ";" ;
135
136 Service_Resource:
137     "resource" resource=[Resource|ID] ";" ;
138
139
140 // FormalParams class
141 FormalParams:
142     "param" name=ID (return?="return")? ";" ;
143
144
145 // Behavior class
146 Behavior:
147     "behavior"
148     "{"
149         (behavior_attrs+=BehaviorAttrs)*
150     "}";
151
152 BehaviorAttrs:
153     Transition |
154     Step;
155
156
157 Transition:
158     from=[Step|ID] "->" to=[Step|ID] ("prob" prob=DOUBLE)?";" ;
159
160
161 // Step class
162 Step:
163     Start |
164     End |
165     Control |
166     Wait |
167     Activity;
168
169
170 // Start class
171 Start:
172     "start" name=ID ";" ;
173
174
175 // End class
176 End:
177     "end" name=ID ";" ;
178
179
180 // Wait class
181 //
182 // name is the name of the wait Step, not the event to wait!
183 // Wait is general (no event specified), in this way the language is more general and flexible
184 // leaving the responsability of binding to the WaitBinding (aka signal) class
185 Wait:
186     "wait" name=ID ";" ;
```

```
187
188
189 // Control class
190 Control:
191     Branch |
192     Fork |
193     Join;
194
195
196 // Branch class
197 Branch:
198     "branch" name=ID ";" ;
199
200
201 // Fork class
202 Fork:
203     "fork" name=ID ";" ;
204
205
206 // Join class
207 Join:
208     "join" name=ID "(" transitionsNeededToGo=INT ")" ";" ;
209
210
211 // intermediate class
212 Activity:
213     SimpleActivity |
214     Reconfiguration |
215     Acquire |
216     Release |
217     ServiceControl;
218
219
220 // Activity class
221 SimpleActivity:
222     "activity" name=ID
223     "{"
224         (activity_attrs+=ActivityAttrs)*
225     "}";
226
227 ActivityAttrs:
228     Activity_InternalExecTime |
229     Activity_InternalFailProb |
230     Activity_InternalFailTime |
231     Activity_Repetitions |
232     Activity_Behavior;
233
234 Activity_InternalExecTime:
235     "execTime" internalExecTime=ProbabilityDistributionFunction ";" ;
236
237 Activity_InternalFailProb:
238     "failProb" internalFailProb=ProbabilityDistributionFunction ";" ;
239
240 Activity_InternalFailTime:
241     "failTime" internalFailTime=ProbabilityDistributionFunction ";" ;
242
243 Activity_Repetitions:
244     "repetitions" repetitions=ProbabilityDistributionFunction ";" ;
245
246 Activity_Behavior:
247     nestedBehavior=Behavior;
248
249
250 // intermediate class
251 Reconfiguration:
252     SimpleReconfiguration |
253     CreateBinding |
254     DeleteBinding;
255
256
```

```
257  /**
258   * For classes below I should have preferred a form like
259   *     activity=SimpleActivity "reconfigure" ...
260   * but the problem is that ANTLR is left valued (see ANTRL manual); that means that it applies
261   * the rules check from left to right and it doesn't make (explicitly) backtracking, so if it
         finds two
262   * rules that start at the same manner it isn't able to resolve the ambiguity and chooses
263   * to drop all the variants except the first one.
264   * So I need the rules start whit a different keyword to make them left valued!!!
265   * This concept applies to:
266   * - SimpleReconfiguration
267   * - CreateBinding
268   * - DeleteBinding
269   * - Acquire
270   * - Release
271   */
272
273
274  // Reconfiguration class
275  SimpleReconfiguration:
276      "reconfigure" name=ID "for" sourceStep=[Step|ID] "to" targetService=[Service|ID] "with"
             activity=SimpleActivity;
277
278
279  // CreateBinding class
280  CreateBinding:
281      "bind" name=ID "for" sourceStep=[Step|ID] "to" targetService=[Service|ID]  "with" activity=
             SimpleActivity;
282
283
284  // DeleteBinding class
285  DeleteBinding:
286      "!" "bind" name=ID "for" sourceStep=[Step|ID] "to" targetService=[Service|ID] "with"
             activity=SimpleActivity;
287
288
289  // Acquire class
290  Acquire:
291      "acquire" name=ID "(" resourceUnit=INT ")" (resource=[Resource|ID])? "with" activity=
             SimpleActivity;
292
293
294  // Release class
295  Release:
296      "release" name=ID "(" resourceUnit=INT ")" (resource=[Resource|ID])? "with" activity=
             SimpleActivity;
297
298
299  // ServiceControl class
300  ServiceControl:
301      "control" name=ID
302      "{"
303          (control_attrs+=ControlAttrs)*
304      "}";
305
306  ControlAttrs:
307       ActivityAttrs |
308       Control_ResourceType |
309       Control_ServiceName |
310       Control_IsSynch |
311       Binding |
312       ActualParam;
313
314  Control_ResourceType:
315      "resourceType" resourceType=STRING ";" ;
316
317  Control_ServiceName:
318      "serviceName" serviceName=[Service|ID] ";" ;
319
320  Control_IsSynch:
```

```
321     "isSynch" isSynch=BooleanType ";" ;
322
323 Enum BooleanType:
324     btrue="true" |
325     bfalse="false";
326
327
328 // Binding class
329 Binding:
330     ServiceBinding |
331     WaitBinding;
332
333 ServiceBinding:
334     "call" call=[Service|ID] ";" ;
335
336 WaitBinding:
337     "signal" signal=[Wait|ID] ";" ;
338
339
340 // ActualParam class
341 ActualParam:
342     "param" name=[FormalParams|ID] "=" value=MathExpr ";" ;
343
344
345
346
347 /**
348  * MathExpr implements a grammar for mathematical expressions composed by
349  * numbers (integer or double), operands (+,-,*,/,^ where * and / have
350  * more precedence than + and - and ^ has more precedence than each other operand)
351  * and FormalParams identifiers.
352  *
353  * example: -1 + 4.5 * (x^6) where x is the identifier of a FormalParam
354  */
355 MathExpr: left=Mexpr (rights+=RightMathExpr)* ;
356
357 RightMathExpr: (op="+"|op="-") right=Mexpr;
358
359 Mexpr: left=Dexpr (rights+=RightMexpr)* ; // Mexpr is at a different level from MathExpr to make
       * and / stronger than + and -
360
361 RightMexpr: (op="*"|op="/") right=Dexpr;
362
363 Dexpr: element=Atom (op="^" pow=Atom)* ; // pow expression
364
365 Atom: Element | ParenthesizedExpr;
366
367 Element: NumericElement | ParamElement;     // an Element can be a number or a reference to a
       FormalParam
368
369 NumericElement: IntNumericElement | DoubleNumericElement;
370
371 IntNumericElement: value=INT;
372
373 DoubleNumericElement: value=DOUBLE;
374
375 ParamElement: value=[FormalParams|ID];
376
377 ParenthesizedExpr: "(" MathExpr ")" ;
378
379
380
381
382 Native DOUBLE :
383     "('-')?('0'..'9')+'.'('0'..'9')+";
```

Appendix B

Use case example

Here we show the KLAPER model of the example used in chapter 10 and expressed using the Domain Specific Language implemented into one of the Eclipse plugins of the KLAPER project (whose EBNF grammar can be found in appendix A). It refers to the starting condition of the example, when any change has not yet been made.

```
model three_tier;

workload system_workload
{
        type OPEN;
//      type CLOSED;
        arrivalProcess exp 0.87;
//      population 20;
        behavior
        {
                start start_ow;
//              activity think_ow
//              {
//                      repetitions const 1.0;
//                      execTime exp 2.0;
//                      failProb const 0.0;
//              }
                control call_ow
                {
                        repetitions const 1.0;
                        execTime const 0.0;
                        failProb const 0.0;
                        isSynch true;
                        call server_query_connector; //client_query;
                }
                end end_ow;

//              start_ow -> think_ow;
//              think_ow -> call_ow;
                start_ow -> call_ow;
                call_ow ->end_ow;
        }
}

//resource client_cpu
//{
```

```
37  //      scheduling RoundRobin;
38  //      type "cpu";
39  //}
40
41  //service process_client_cpu
42  //{
43  //      resource client_cpu;
44  //      param num_opts_cl;
45  //      behavior
46  //      {
47  //              start start_cl_cpu;
48  //              activity act_cl_cpu
49  //              {
50  //                      repetitions const 1.0;
51  //                      execTime exp 0.000000000738;
52  //                      failProb const 0.05;
53  //              }
54  //              end end_cl_cpu;
55  //
56  //              start_cl_cpu -> act_cl_cpu;
57  //              act_cl_cpu -> end_cl_cpu;
58  //      }
59  //}
60
61  //resource client
62  //{
63  //      scheduling RoundRobin;
64  //      type "application";
65  //}
66
67  //service client_query
68  //{
69  //      resource client;
70  //      behavior
71  //      {
72  //              start start_cl;
73  //              control compute_cl
74  //              {
75  //                      repetitions const 1.0;
76  //                      execTime const 0.0;
77  //                      failProb const 0.0;
78  //                      isSynch true;
79  //                      call process_client_cpu;
80  //                      param num_opts_cl=1.0;
81  //              }
82  //              control query_cl
83  //              {
84  //                      repetitions const 1.0;
85  //                      execTime const 0.0;
86  //                      failProb const 0.0;
87  //                      isSynch true;
88  //                      call server_query_connector;
89  //              }
90  //              end end_cl;
91  //
92  //              start_cl -> compute_cl;
93  //              compute_cl -> query_cl;
94  //              query_cl -> end_cl;
95  //      }
96  //}
97
98  resource client_net
99  {
100         scheduling RoundRobin;
101         type "network";
102         capacity 125.0;
103 }
104
105 service transmit_cl_net
106 {
```

```
107         resource client_net;
108         param num_bytes_cl;
109         behavior
110         {
111                 start start_cl_net;
112                 activity act_cl_net
113                 {
114                         repetitions const 1.0;
115                         execTime exp 0.008192; // 100KB * 0.00000008;
116                         failProb const 0.007;
117                 }
118                 end end_cl_net;
119
120                 start_cl_net -> act_cl_net;
121                 act_cl_net -> end_cl_net;
122         }
123 }
124
125 resource server_conn
126 {
127         scheduling RoundRobin;
128         type "connector";
129         capacity 32760.0;
130 }
131
132 service server_query_connector
133 {
134         resource server_conn;
135         behavior
136         {
137                 start start_srv_conn;
138                 control send_srv_conn
139                 {
140                         repetitions const 1.0;
141                         execTime const 0.0;
142                         failProb const 0.0;
143                         isSynch true;
144                         call transmit_cl_net;
145                         param num_bytes_cl=1.0;
146                 }
147                 control server_query_conn
148                 {
149                         repetitions const 1.0;
150                         execTime const 0.0;
151                         failProb const 0.0;
152                         isSynch true;
153                         call server_query;
154                 }
155                 control recv_srv_conn
156                 {
157                         repetitions const 1.0;
158                         execTime const 0.0;
159                         failProb const 0.0;
160                         isSynch true;
161                         call transmit_cl_net;
162                         param num_bytes_cl=1.0;
163                 }
164                 end end_srv_conn;
165
166                 start_srv_conn -> send_srv_conn;
167                 send_srv_conn -> server_query_conn;
168                 server_query_conn -> recv_srv_conn;
169                 recv_srv_conn -> end_srv_conn;
170         }
171 }
172
173 resource server_cpu
174 {
175         scheduling RoundRobin;
176         type "cpu";
```

```
177 }
178
179 service process_server_cpu
180 {
181         resource server_cpu;
182         param num_opts_srv;
183         behavior
184         {
185                 start start_srv_cpu;
186                 activity act_srv_cpu
187                 {
188                         repetitions const 1.0;
189                         execTime exp 0.1; //0.000000000738;
190                         failProb const 0.0001;
191                 }
192                 end end_srv_cpu;
193
194                 start_srv_cpu -> act_srv_cpu;
195                 act_srv_cpu -> end_srv_cpu;
196         }
197 }
198
199 resource server
200 {
201         scheduling RoundRobin;
202         type "application";
203         capacity 32760.0;
204 }
205
206 service server_query
207 {
208         resource server;
209         behavior
210         {
211                 start start_srv;
212                 control compute_srv
213                 {
214                         repetitions const 1.0;
215                         execTime const 0.0;
216                         failProb const 0.0;
217                         isSynch true;
218                         call process_server_cpu;
219                         param num_opts_srv=1.0;
220                 }
221                 control query_srv
222                 {
223                         repetitions const 1.0;
224                         execTime const 0.0;
225                         failProb const 0.0;
226                         isSynch true;
227                         call db_query_connector;
228                 }
229                 end end_srv;
230
231                 start_srv -> compute_srv;
232                 compute_srv -> query_srv;
233                 query_srv -> end_srv;
234         }
235 }
236
237 resource server_net
238 {
239         scheduling RoundRobin;
240         type "network";
241         capacity 62.5;
242 }
243
244 service transmit_srv_net
245 {
246         resource server_net;
```

```
247         param num_bytes_srv;
248         behavior
249         {
250                 start start_srv_net;
251                 activity act_srv_net
252                 {
253                         repetitions const 1.0;
254                         execTime exp 0.016384; // 200KB * 0.00000008;
255                         failProb const 0.10;  // <--- reliability (step1: 0.10, step2: 0.007,
                                step3: 0.007)
256                 }
257                 end end_srv_net;
258 
259                 start_srv_net -> act_srv_net;
260                 act_srv_net -> end_srv_net;
261         }
262 }
263 
264 resource db_conn
265 {
266         scheduling RoundRobin;
267         type "connector";
268         capacity 32760.0;
269 }
270 
271 service db_query_connector
272 {
273         resource db_conn;
274         behavior
275         {
276                 start start_db_conn;
277                 control send_db_conn
278                 {
279                         repetitions const 1.0;
280                         execTime const 0.0;
281                         failProb const 0.0;
282                         isSynch true;
283                         call transmit_srv_net;
284                         param num_bytes_srv=1.0;
285                 }
286                 control db_query_conn
287                 {
288                         repetitions const 1.0;
289                         execTime const 0.0;
290                         failProb const 0.0;
291                         isSynch true;
292                         call db_query;
293                 }
294                 control recv_db_conn
295                 {
296                         repetitions const 1.0;
297                         execTime const 0.0;
298                         failProb const 0.0;
299                         isSynch true;
300                         call transmit_srv_net;
301                         param num_bytes_srv=1.0;
302                 }
303                 end end_db_conn;
304 
305                 start_db_conn -> send_db_conn;
306                 send_db_conn -> db_query_conn;
307                 db_query_conn -> recv_db_conn;
308                 recv_db_conn -> end_db_conn;
309         }
310 }
311 
312 resource db_cpu
313 {
314         scheduling RoundRobin;
315         type "cpu";
```

```
316 }
317
318 service process_db_cpu
319 {
320         resource db_cpu;
321         param num_opts_db;
322         behavior
323         {
324                 start start_db_cpu;
325                 activity act_db_cpu
326                 {
327                         repetitions const 1.0;
328                         execTime exp 0.5; //0.000000000738;
329                         failProb const 0.0001;
330                 }
331                 end end_db_cpu;
332
333                 start_db_cpu -> act_db_cpu;
334                 act_db_cpu -> end_db_cpu;
335         }
336 }
337
338 resource db
339 {
340         scheduling RoundRobin;
341         type "application";
342         capacity 32760.0;
343 }
344
345 service db_query
346 {
347         resource db;
348         behavior
349         {
350                 start start_db;
351                 control compute_db
352                 {
353                         repetitions const 1.0;
354                         execTime const 0.0;
355                         failProb const 0.0;
356                         isSynch true;
357                         call process_db_cpu;
358                         param num_opts_db=1.0;
359                 }
360                 control disk_io_db
361                 {
362                         repetitions const 1.0;
363                         execTime const 0.0;
364                         failProb const 0.0;
365                         isSynch true;
366                         call disk_io;
367                 }
368                 end end_db;
369
370                 start_db -> compute_db;
371                 compute_db -> disk_io_db;
372                 disk_io_db -> end_db;
373         }
374 }
375
376 resource disk
377 {
378         scheduling RoundRobin;
379         type "disk";
380         capacity 1.0;
381 }
382
383 service disk_access
384 {
385         resource disk;
```

```
386         param num_bytes_disk;
387         behavior
388         {
389                 start start_disk;
390                 activity act_disk
391                 {
392                         repetitions const 1.0;
393                         execTime exp 1.0; //0.00000000333; // <-- performance (step1: 1.0, step2
                                : 1.0, step3: 0.5)
394                         failProb const 0.02;
395                 }
396                 end end_disk;
397 
398                 start_disk -> act_disk;
399                 act_disk -> end_disk;
400         }
401 }
402 
403 resource db_disk
404 {
405         scheduling RoundRobin;
406         type "dummy_resource";
407         capacity 32760.0;
408 }
409 
410 service disk_io
411 {
412         resource db_disk;
413         behavior
414         {
415                 start start_io;
416                 control call_io
417                 {
418                         repetitions const 1.0;
419                         execTime const 0.0;
420                         failProb const 0.0;
421                         isSynch true;
422                         call disk_access;
423                         param num_bytes_disk=1.0;
424                 }
425                 end end_io;
426 
427                 start_io -> call_io;
428                 call_io -> end_io;
429         }
430 }
```

Appendix C

Transformation Rules from KLAPER to DTMC

Here we report all the transformation rules used to go from the KLAPER meta model to the DTMC meta model.

```
1  // Import source metamodel
2  import klaper::core;
3  import klaper::probability;
4  import klaper::expr;
5
6  //Import target metamodel
7  import dtmc::core;
8
9  //Include a few utility extensions
10 extension org::klaper::util::dsl::Extensions;
11 extension org::klaper::util::ProbExtensions;
12 extension org::klaper::dtmc::m2m::util::klaperToklaper;
13 extension org::klaper::dtmc::m2m::util::dtmcEntities;
14 extension org::klaper::dtmc::m2m::util::forkUtil;
15 extension org::klaper::dtmc::m2m::util::repetitionsUtil;
16 extension org::klaper::dtmc::m2m::util::utils;
17
18 //Include the io extensions, for debugging purposes (in case we need it)
19 extension org::openarchitectureware::util::stdlib::io;
20 extension org::openarchitectureware::util::stdlib::issues;
21 extension org::openarchitectureware::util::stdlib::counter;
22
23 /*
24         Starting point for the transformation (this extension is invoked directly from the
               workflow).
25         PLEASE NOTE: The input KlaperModel is supposed to be well formed for the purpose of
               reliability analisys with DTMC!
26 */
27 dtmc::core::ReliabilityModel klaper2dtmc(klaper::core::KlaperModel m):
28         m.transformModel();
29
30 /*
31         Creates a ReliabilityModel from the input KlaperModel
32 */
33 private create dtmc::core::ReliabilityModel newModel transformModel(klaper::core::KlaperModel m)
       :
34         newModel.dtmc.addAll(m.resource.offeredService.transformService(newModel))->
```

```
35          m.resource.offeredService.behavior.step.typeSelect(klaper::core::ServiceControl).select(
                e|e.binding.signal!=null).collect(e|e.linkWait(newModel))->
36          newModel;
37
38
39  private Void linkWait(klaper::core::ServiceControl s, dtmc::core::ReliabilityModel m):
40                  let extRef=new ExternalReference:
41                  let waitSt=s.binding.signal.retrieveWaitState(m):
42                  let scSt=s.retrieveServiceControlState(m):
43                  extRef.setDependsOn((dtmc::core::DTMC)scSt.eContainer)->
44                  extRef.setNavigateUntil(scSt)->
45                  waitSt.externalReference.add(extRef);
46  /*
47          Creates a DTMC from each Service offered by a given Klaper Resource (Workloads are not
                useful for reliability analsisys).
48          A Fail state is added to the Markov Chain at once.
49  */
50  private create dtmc::core::DTMC newDtmc transformService(klaper::core::Service s, dtmc::core::
        ReliabilityModel m):
51          let f = new Fail:
52          f.setName("Fail")->
53          newDtmc.setName( ((klaper::core::Resource)s.eContainer).name+"_"+s.name )->
54          newDtmc.state.add(f)->
55          newDtmc.state.addAll(s.behavior.step.collect(e|transformStep(e,newDtmc,m)))->
56          newDtmc.state.select(e|e.externalReference.size==0).collect(e|
                updateTransitionProbabilities(e))->
57          newDtmc.state.removeAll(newDtmc.state.select(e|e.name==null))->
58          newDtmc;
59
60  /*******************************************************
61
62  Polymorphic extension transformStep
63
64  *******************************************************/
65
66  /*
67          Dummy transformation for abstract class klaper::core::Step
68  */
69  private create dtmc::core::State newState transformStep(klaper::core::Step s, dtmc::core::DTMC d
        , dtmc::core::ReliabilityModel m):
70          {};
71
72  /*
73          Creates a DTMC Start state from a Klaper Start step
74  */
75  private create dtmc::core::Start newState transformStep(klaper::core::Start s, dtmc::core::DTMC
        d, dtmc::core::ReliabilityModel m):
76          newState.setName(s.name)->
77          d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.select(x|x.from==s
                ).collect(e|e.transformTransition(d,m)))->
78          newState;
79
80  /*
81          Creates a DTMC End state from a Klaper End step
82  */
83  private create dtmc::core::End newState transformStep(klaper::core::End s, dtmc::core::DTMC d,
        dtmc::core::ReliabilityModel m):
84          newState.setName(s.name)->
85          d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.select(x|x.to==s
                && x.from.metaType!=Join).collect(e|e.transformTransition(d,m)))->
86          newState;
87
88
89  /*
90          Creates a DTMC State from a Klaper Activity. A transition towards the Fail state (whose
                probability is elicited from the Activity's internalFailProb attribute)
91          is added to the Chain only in the case the Activity is repeated only once and does not
                contain a nested Behavior. In all the other cases, the Activity is mapped into
92          a new DTMC (by means of transformActivity()) and the State created here contains an
                ExternalReference pointing to such DTMC.
```

```
93  */
94   private create dtmc::core::State newState transformStep(klaper::core::Activity s, dtmc::core::
        DTMC d,  dtmc::core::ReliabilityModel m):
95
96          (s.stepBelongsToForkJoin(0,{})==false)?
97                  (
98                          s.transformActivityRepetitions()->
99                          s.isASimpleActivity()?
100                         (
101                                 newState.setName("ACT_"+s.name)->
102                                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                                        transition.select(x|(x.to==s&& x.from.metaType!=Join)||x.
                                        from==s).collect(e|e.transformTransition(d,m)))->
103                                 d.transition.add(s.transitionToFail(d,m))->
104                                 newState
105                         )
106                         :
107                         (
108                                 m.dtmc.add(s.transformActivity(m))->
109                                 (
110                                         let extRef=new ExternalReference:
111                                         extRef.setDependsOn(s.transformActivity(m))->
112                                         newState.setInternalFailProb(0.0)->
113                                         newState.externalReference.add(extRef)
114                                 )
115                                 ->
116                                 newState.setCompletionModel("OR")->
117                                 newState.setName(s.name)->
118                                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                                        transition.select(x|(x.to==s&& x.from.metaType!=Join)||x.
                                        from==s).collect(e|e.transformTransition(d,m)))->
119                                 newState
120                         )
121                 )
122                 :{};
123
124 /*
125         Creates a DTMC State from a KLAPER Wait step
126 */
127 private create dtmc::core::State newState transformStep(klaper::core::Wait s, dtmc::core::DTMC d
        ,  dtmc::core::ReliabilityModel m):
128         (s.stepBelongsToForkJoin(0,{})==false)?
129         (
130                 newState.setName(s.name)->
131                 newState.setCompletionModel("OR")->
132                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.select(x|(
                        x.to==s&& x.from.metaType!=Join)||x.from==s).collect(e|e.transformTransition
                        (d,m)))->
133                 newState
134         ):
135         {};
136
137 /*
138         Creates a DTMC State from a Klaper ServiceControl. In the case the ServiceControl is
                repeated just once, newState contains already an ExternalReference pointing to
139         the DTMC representing the invoked service and its internalFailProb is set with the same
                value as the original ServiceControl. In all the other cases,
140         the ServiceControl is mapped into a new DTMC (by means of transformServiceControl()) and
                the State created here contains an ExternalReference pointing to such DTMC.
141 */
142 private create dtmc::core::State newState transformStep(klaper::core::ServiceControl s, dtmc::
        core::DTMC d,  dtmc::core::ReliabilityModel m):
143         (s.stepBelongsToForkJoin(0,{})==false)?
144                 (//belongsToForkJoin=false
145                         s.transformActivityRepetitions()->
146
147                         s.isASignalServiceControl()?
148                         (//signal
149
150                                 newState.setName( s.name )->
```

```
151                                     newState.setInternalFailProb(s.internalFailProb.getMean())->
152                                     d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                                             transition.select(x|(x.to==s&& x.from.metaType!=Join)||x.
                                             from==s).collect(e|e.transformTransition(d,m)))->
153                                     d.transition.add(s.transitionToFail(d,m))->
154                                     newState
155                           )//signal
156                           :
157                           (//call
158 
159                            (s.isSynch==false&&s.dependsOn==false)?
160                            (//synch=false
161                                     newState.setName(s.name)->
162                                     d.transition.addAll(((klaper::core::Behavior)s.eContainer).
                                             transition.select(x|(x.to==s&& x.from.metaType!=Join)||x.
                                             from==s).collect(e|e.transformTransition(d,m)))->
163                                     d.transition.add(s.transitionToFail(d,m))->
164                                     newState
165                            )//synch=false
166                            :
167                            (//synch==true
168                                     s.isASimpleServiceControl()?
169                                     (//simpleSC
170                                              newState.setName("SC_"+s.name)->
171                                              (
172                                                       let extRef=new ExternalReference:
173                                                       extRef.setDependsOn(s.binding.call.
                                                                transformService(m))->
174                                                       newState.setInternalFailProb(s.internalFailProb.
                                                                getMean())->
175                                                       newState.externalReference.add(extRef)->
176                                                       newState.setCompletionModel("OR")
177                                              )->
178                                              d.transition.addAll(((klaper::core::Behavior)s.
                                                   eContainer).transition.select(x|(x.to==s&& x.from.
                                                   metaType!=Join)||x.from==s).collect(e|e.
                                                   transformTransition(d,m)))->
179                                              newState
180                                     )//simpleSC
181                                     :
182                                     (//complexSC
183                                              m.dtmc.add(s.transformServiceControl(m))->
184 
185                                              (
186                                                       let extRef=new ExternalReference:
187                                                       extRef.setDependsOn(s.transformServiceControl(m)
                                                                )->
188                                                       newState.setInternalFailProb(0.0)->
189                                                       newState.externalReference.add(extRef)
190                                              )
191                                              ->
192                                              newState.setCompletionModel("OR")->
193                                              newState.setName(s.name)->
194                                              d.transition.addAll(((klaper::core::Behavior)s.
                                                   eContainer).transition.select(x|(x.to==s&& x.from.
                                                   metaType!=Join)||x.from==s).collect(e|e.
                                                   transformTransition(d,m)))->
195                                              newState
196                                     )//complexSC
197                           )//synch=true
198                           )//call
199                  )//belongsToForkJoin=false
200                  :
201                  {};
202 
203 /*
204         A top-level Fork (i.e. a Fork which is not nested in another Fork-Join pattern) is
                 mapped into a new State which points (by means of n ExternalReferences)
205         to the DTMCs corresponding to the n paths starting from such Fork. Each path is first
                 transformed into a Klaper Service (by means of transformForkPathToService())
```

```
206         and then such Service is transformed into a DTMC (createExternalReference() does this
                job).
207
208   */
209   private create dtmc::core::State newState transformStep(klaper::core::Fork s, dtmc::core::DTMC d
          , dtmc::core::ReliabilityModel m):
210         (s.stepBelongsToForkJoin(0,{})==false)?
211         (
212                 let pathServices = (List[klaper::core::Service]){}://Each path starting from
                        Fork is mapped to a Service
213                 s.out.collect(e|pathServices.add(e.to.transformForkPathToService(s,m)))->
214
215                 newState.externalReference.addAll(pathServices.collect(e|e.
                        createExternalReference(m)))->
216                 newState.setName(s.name)->
217                 newState.setCompletionModel(s.retrieveJoinFromFork(0,{}).in.size==s.
                        retrieveJoinFromFork(0,{}).transitionsNeededToGo?"AND":"OR")->
218                 d.state.add(newState)->
219                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.select(e|e
                        .to==s).collect(e|e.transformTransition(d,m)))->
220                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.select(e|e
                        .from==s.retrieveJoinFromFork(0,{})).collect(e|e.transformJoinTransition(d,
                        newState,m)))
221         ):
222         {};
223
224
225   /*
226         Creates a DTMC State corresponding to a Klaper Branch Step.
227   */
228   private create dtmc::core::State newState transformStep(klaper::core::Branch s, dtmc::core::DTMC
        d, dtmc::core::ReliabilityModel m):
229
230         (s.stepBelongsToForkJoin(0,{})==false)?
231         (
232                 newState.setName(s.name)->
233                 d.transition.addAll(((klaper::core::Behavior)s.eContainer).transition.select(x|(
                        x.to==s&& x.from.metaType!=Join)||x.from==s).collect(e|e.transformTransition
                        (d,m)))->
234                 newState
235         ):
236         {};
237
238   /*******************************************************
239
240   END of Polymorphic extension transformStep
241
242   *******************************************************/
243
244   /*
245         Creates a DTMC from a Klaper Behavior (this can be a Behavior nested in an Activity, as
                well as the Behavior of a fictitious Service created for a Fork path).
246         A Fail state is added to the Markov Chain at once.
247   */
248   private create dtmc::core::DTMC newDtmc transformBehavior(klaper::core::Behavior b, dtmc::core::
        ReliabilityModel m):
249         let f=new Fail:
250         f.setName("Fail")->
251         newDtmc.setName(((klaper::core::Activity)(b.eContainer)).name+"_BEHAV")->
252         newDtmc.state.add(f)->
253         newDtmc.state.addAll(b.step.collect(e|transformStep(e,newDtmc,m)))->
254         newDtmc.state.select(e|e.externalReference.size==0).collect(e|
                updateTransitionProbabilities(e))->
255         newDtmc.state.removeAll(newDtmc.state.select(e|e.name==null))->
256         newDtmc;
257
258   /*
259         Creates a dummy Klaper Service for a path having firstStep as its 1st Step and starting
                from f.
260   */
```

```
261 private create klaper::core::Service newServ transformForkPathToService (klaper::core::Step
        firstStep , klaper::core::Fork f, dtmc::core::ReliabilityModel m):
262         let newB=new klaper::core::Behavior:
263         newServ.setName(f.name+"_PATH_TO_"+firstStep.name)->
264         f.copyForkPath(firstStep ,newB,1,{})->
265         newServ.setBehavior(newB)->
266         newServ;
267 
268 
269 /*
270         Transforms a Klaper Activity into a DTMC. The structure of such DTMC depends on the
            distribution function used for "repetitions" attribute of Activity.
271         In the case an Histogram function is used, this structure is variable and
            createActivityStateList() is invoked in order to do the job.
272         Otherwise, the structure is fixed and the job is done directly within this extension.
273 */
274 private create dtmc::core::DTMC newDtmc transformActivity(klaper::core::Activity a, dtmc::core::
        ReliabilityModel m):
275 
276 ((klaper::core::Service)((klaper::core::Behavior)a.eContainer).eContainer).name.contains("
        PATH_TO")?
277 newDtmc.setName("ACT_"+((klaper::core::Service)((klaper::core::Behavior)a.eContainer).eContainer
        ).name.split("_PATH_TO").first()+"_"+a.name):
278         newDtmc.setName( "ACT_"+((klaper::core::Service)((klaper::core::Behavior)a.eContainer).
            eContainer).name+"_"+a.name )->
279 
280         newDtmc.state.add(createFailState("Fail",a))->
281         newDtmc.state.add(createStartState("START",a))->
282         newDtmc.state.add(createEndState("End",a))->
283         newDtmc.state.add(createState("Switch",a))->
284 
285         a.repetitions.metaType==Geometric?
286         (
287                 a.nestedBehavior!=null?
288                 (
289                         let extRef=new ExternalReference:
290                         m.dtmc.add(transformBehavior(a.nestedBehavior,m))->
291                         extRef.setDependsOn(transformBehavior(a.nestedBehavior,m))->
292                         createState(a.name,a).setInternalFailProb(0.0)->
293                         createState(a.name,a).setCompletionModel("OR")->
294                         createState(a.name,a).externalReference.add(extRef)
295                 ):
296                 {}      ->
297                 newDtmc.state.add(createState(a.name,a))->
298                 newDtmc.transition.add(createTransition(createStartState("START",a),createState(
                    "Switch",a),1.0,a))->
299                 newDtmc.transition.add(createTransition(createState("Switch",a),createState(a.
                    name,a),1.0-a.repetitions.getSuccessProbability(),a))->
300                 newDtmc.transition.add(createTransition(createState("Switch",a),createEndState("
                    End",a),a.repetitions.getSuccessProbability(),a))->
301                 newDtmc.transition.add(createTransition(createState(a.name,a),createState("
                    Switch",a),1.0-a.internalFailProb.getMean(),a))->
302                 newDtmc.transition.add(createTransition(createState(a.name,a),createFailState("
                    Fail",a),a.internalFailProb.getMean(),a))
303         )
304         :
305         (
306                 a.repetitions.metaType==Histogram?
307                 (
308                         newDtmc.transition.add(createTransition(createStartState("START",a),
                            createState("Switch",a),1.0,a))->
309                         ((klaper::probability::Histogram)(a.repetitions)).samples.collect(e|
310                                 createActivityStateList(e.value,1,createFailState("Fail",a),
                                    createState("Switch",a),createEndState("End",a),newDtmc,e.
                                    probability,a,m))
311                 )
312                 :
313                 (
314                 {}
315                 )
```

```
316
317         )
318         ->
319         newDtmc.state.removeAll(newDtmc.state.select(e|e.name==null))->
320         newDtmc;
321
322 /*
323         Support extension for transformActivity(). Transforms each Sample of an Histogram
            representing the number of repetitions for a given Activity
324         into a path of listLen States each representing a single repetition of a.
325 */
326 private create dtmc::core::State newState createActivityStateList(Integer listLen, Integer count
    , dtmc::core::Fail f, dtmc::core::State swi,
327                              dtmc::core::End end, dtmc::core::DTMC d, Real listProb, klaper::core
                                   ::Activity a, dtmc::core::ReliabilityModel m):
328         listLen==0?
329         (
330                 d.transition.add(createTransition (swi,end,listProb,a))
331         )
332         :
333         (
334
335                 a.nestedBehavior!=null?
336                 (
337
338                         let extRef=new ExternalReference:
339                         m.dtmc.add(transformBehavior(a.nestedBehavior,m))->
340                         extRef.setDependsOn(transformBehavior(a.nestedBehavior,m))->
341                         newState.setInternalFailProb(0.0)->
342                         newState.setCompletionModel("OR")->
343                         newState.externalReference.add(extRef)
344                 ):
345                 {}      ->
346                 newState.setName("REP"+count+" of"+listLen)->
347                 d.state.add(newState)->
348                 a.nestedBehavior==null?
349                 d.transition.add(createTransition(newState,f,a.internalFailProb.getMean(),a))
                        :{}->
350                 count==1?
351                 (
352                         d.transition.add(createTransition (swi,newState,listProb,a))
353                 )
354                 :
355                 {}
356                 ->
357                 count==listLen?
358                 (
359                 a.nestedBehavior==null?
360                         d.transition.add(createTransition (newState,end,1.0-a.internalFailProb.
                                getMean(),a)):
361                         d.transition.add(createTransition (newState,end,1.0,a))
362                 )
363                 :
364                 (
365
366                         d.transition.add(createTransition(newState,createActivityStateList(
                                listLen,count+1,f,swi,end,d,listProb,a,m),
367                         a.nestedBehavior==null?
368                         1.0-a.internalFailProb.getMean():1.0,a)
369                 )
370         ))
371         ;
372
373 /*
374         Transforms a Klaper ServiceControl into a DTMC. The structure of such DTMC depends on
            the distribution function used for "repetitions" attribute of ServiceControl.
375         In the case an Histogram function is used, this structure is variable and
            createServiceControlStateList() is invoked in order to do the job.
376         Otherwise, the structure is fixed and the job is done directly within this extension.
377 */
```

```
378 private create dtmc::core::DTMC newDtmc transformServiceControl (klaper::core::ServiceControl a,
      dtmc::core::ReliabilityModel m):
379
380 ((klaper::core::Service)((klaper::core::Behavior)a.eContainer).eContainer).name.contains("
     PATH_TO")?
381 newDtmc.setName("SC_"+((klaper::core::Service)((klaper::core::Behavior)a.eContainer).eContainer)
     .name.split("_PATH_TO").first()+"_"+a.name):
382          newDtmc.setName( "SC_"+((klaper::core::Service)((klaper::core::Behavior)a.eContainer).
                eContainer).name+"_"+a.name )->
383
384          newDtmc.state.add(createFailState("Fail",a))->
385          newDtmc.state.add(createStartState("START",a))->
386          newDtmc.state.add(createEndState("End",a))->
387          newDtmc.state.add(createState("Switch",a))->
388
389          a.repetitions.metaType==Geometric?
390          (
391
392                   let extRef=new ExternalReference:
393                   extRef.setDependsOn(a.binding.call.transformService(m))->
394                   createState(a.name,a).setInternalFailProb(a.internalFailProb.getMean())->
395                   createState(a.name,a).setCompletionModel("OR")->
396                   createState(a.name,a).externalReference.add(extRef)->
397                   newDtmc.state.add(createState(a.name,a))->
398
399                   newDtmc.transition.add(createTransition(createStartState("START",a),createState(
                         "Switch",a),1.0,a))->
400                   newDtmc.transition.add(createTransition(createState("Switch",a),createState(a.
                         name,a),1.0-a.repetitions.getSuccessProbability(),a))->
401                   newDtmc.transition.add(createTransition(createState("Switch",a),createEndState("
                         End",a),a.repetitions.getSuccessProbability(),a))->
402                   newDtmc.transition.add(createTransition(createState(a.name,a),createState("
                         Switch",a),1.0,a))
403          )
404          :
405          (
406                   a.repetitions.metaType==Histogram?
407                   (
408                            newDtmc.transition.add(createTransition(createStartState("START",a),
                                  createState("Switch",a),1.0,a))->
409                            ((klaper::probability::Histogram)(a.repetitions)).samples.collect(e|
410                                     createServiceControlStateList(e.value,1,createFailState("Fail",a
                                          ),createState("Switch",a),createEndState("End",a),newDtmc,e.
                                          probability,a,m))
411                   )
412                   :
413                   (
414                   {}
415                   )
416
417          )
418          ->
419          newDtmc.state.removeAll(newDtmc.state.select(e|e.name==null))->
420          newDtmc;
421
422 /*
423          Support extension for transformServiceControl(). Transforms each Sample of an Histogram
                representing the number of repetitions for a given ServiceControl
424          into a path of listLen States each representing a single repetition of a.
425 */
426 private create dtmc::core::State newState createServiceControlStateList(Integer listLen, Integer
      count, dtmc::core::Fail f, dtmc::core::State swi,
427                              dtmc::core::End end, dtmc::core::DTMC d, Real listProb, klaper::core
                                   ::ServiceControl a, dtmc::core::ReliabilityModel m):
428          listLen==0?
429          (
430                   d.transition.add(createTransition (swi,end,listProb,a))
431          )
432          :
433          (
```

```
434                 let extRef=new ExternalReference:
435                 extRef.setDependsOn(a.binding.call.transformService(m))->
436                 newState.setInternalFailProb(a.internalFailProb.getMean())->
437                 newState.setCompletionModel("OR")->
438                 newState.externalReference.add(extRef)->
439                 newState.setName("REP"+count+"of"+listLen)->
440                 d.state.add(newState)->
441 
442                 count==1?
443                 (
444                         d.transition.add(createTransition (swi,newState,listProb,a))
445                 )
446                 :
447                 {}
448                 ->
449                 count==listLen?
450                 (
451                         d.transition.add(createTransition (newState,end,1.0,a))
452                 )
453                 :
454                 (
455                         d.transition.add(createTransition(newState,createServiceControlStateList
                             (listLen,count+1,f,swi,end,d,listProb,a,m),1.0,a)
456                 )
457         ))
458         ;
459 
460 
461 /*
462         Creates an ExternalReference "pointing" to the DTMC corresponding to Service s' Behavior
             .
463 */
464 
465 private create dtmc::core::ExternalReference newRef createExternalReference(klaper::core::
     Service s, dtmc::core::ReliabilityModel m):
466         m.dtmc.add(s.behavior.transformBehavior(m))->
467         newRef.setDependsOn(s.behavior.transformBehavior(m))->
468         newRef;
469 
470 /*
471         Create a DTMC Transition corresponding to a Klaper Transition which doesn't start from a
             Join step.
472 */
473 private create dtmc::core::Transition newTrans transformTransition(klaper::core::Transition t,
     dtmc::core::DTMC d, dtmc::core::ReliabilityModel m):
474         t.from.metaType==Join?transformJoinTransition(t,d,transformStep(t.from.
             retrieveForkFromJoin(0,{}),d,m),m):
475         (
476         newTrans.setName(t.from.name+"_to_"+t.to.name)->
477         newTrans.setFrom(transformStep(t.from,d,m))->
478         newTrans.setTo(transformStep(t.to, d,m) )->
479         newTrans.setProbability(t.prob)->
480         newTrans);
481 
482 
483 /*
484          Create a DTMC Transition corresponding to a Klaper Transition which starts from a Join
             step.
485 */
486 private create dtmc::core::Transition newTrans transformJoinTransition(klaper::core::Transition
     t, dtmc::core::DTMC d, dtmc::core::State fromState, dtmc::core::ReliabilityModel m):
487         newTrans.setName(fromState.name+"_to_"+transformStep(t.to,d,m).name)->
488         newTrans.setFrom(fromState)->
489         newTrans.setTo(transformStep(t.to, d,m) )->
490         newTrans.setProbability(t.prob)->
491         newTrans;
492 
493 /*
494         Create a DTMC Transition towards a DTMC Fail State
495 */
```

```
496 private create dtmc::core::Transition newTrans transitionToFail(klaper::core::Activity s, dtmc::
        core::DTMC d, dtmc::core::ReliabilityModel m):
497                 s.internalFailProb!=null?
498                 (
499                 newTrans.setName(s.name+"_to_Fail")->
500                 newTrans.setFrom(transformStep(s,d,m))->
501                 newTrans.setTo(d.state.selectFirst(e|e.metaType==Fail))->
502                 newTrans.setProbability(s.internalFailProb.getMean())->
503                 newTrans
504                 ):
505                 null;
506
507 /*
508         Update transition probabilities starting from s by taking into account the probability
                of the transitions towards Fail state.
509 */
510 private updateTransitionProbabilities(dtmc::core::State s):
511         {
512                 let p_fail=s.outgoing.select(e|e.to.metaType==Fail).size==1? s.outgoing.
                        selectFirst(e|e.to.metaType==Fail).probability:0.0:
513                 s.outgoing.select(e|e.to.metaType!=Fail).collect(x|x.setProbability(x.
                        probability-p_fail))
514         };
```

Appendix D

Transformation Rules from KLAPER to SimJava

Here we report all the transformation rules used to go from the KLAPER meta model to the SimJava meta model.

```
1  import klaper::core;
2  import klaper::probability;
3  import klaper::expr;
4
5  import simulator::core;
6  import simulator::probability;
7
8  // to include the toDouble() extension
9  extension org::klaper::util::dsl::Extensions;
10
11 // to include probability distribution functions
12 extension org::klaper::util::ProbExtensions;
13
14 // and we also load the io extensions, for the purpose of
15 // debugging, in case we need it
16 extension org::openarchitectureware::util::stdlib::io;
17 extension org::openarchitectureware::util::stdlib::issues;
18 extension org::openarchitectureware::util::stdlib::counter;
19
20 // starting point for the transformation
21 // (this extension is called from the workflow!)
22 simulator::core::SimModel klaper2sim(KlaperModel m):
23       m.transformModel();
24
25 // Klaper KlaperModel class
26 private create simulator::core::SimModel this transformModel(klaper::core::KlaperModel m):
27     this.setResource(m.resource.transformResource())->
28 //       this.setWorkload(m.workload.select(w|w.type.toString()=='OPEN').transformOpenWorkload())
       ->
29           this.workload.addAll(m.workload.select(w|w.type.toString()=='OPEN').
               transformOpenWorkload())->
30 //   this.setWorkload(m.workload.select(w|w.type.toString()=='CLOSED').transformClosedWorkload())
       ->
31     this.workload.addAll(m.workload.select(w|w.type.toString()=='CLOSED').
          transformClosedWorkload())->
32     this;
33
```

```
34 private create simulator::core::OpenWorkload this transformOpenWorkload(klaper::core::Workload w
       ):
35         this.setName(w.name)->
36         this.setStep(w.behavior.step.transformStep())->
37         this.setArrivalProcess(w.arrivalProcess.transformDistribution())->
38         this;
39
40 private create simulator::core::ClosedWorkload this transformClosedWorkload(klaper::core::
       Workload w):
41         this.setPopulation(w.population)->
42         this.setName(w.name)->
43         this.setStep(w.behavior.step.transformStep())->
44         this;
45
46 private create simulator::core::ResourceQueue this transformResource(klaper::core::Resource r):
47         this.setCapacity(r.capacity.toInteger())->
48         this.setDescription(r.description)->
49         this.setName(r.name)->
50         this.setOfferedService(r.offeredService.transformService())->
51         this;
52
53 private create simulator::core::Service this transformService(klaper::core::Service s):
54         this.setName(s.name)->
55         this.setDescription(s.description)->
56         this.setStep(s.behavior.step.transformStep())->
57         this;
58
59 private create simulator::core::Step this transformStep(klaper::core::Step s):
60         this;
61
62 private create simulator::core::Start this transformStep(klaper::core::Start s):
63         this.setName(s.name)->
64         this.setOutTransition(((klaper::core::Behavior)s.eContainer).transition.select(t|t.from.
             name==s.name).transformTransition())->
65         this;
66
67 private create simulator::core::End this transformStep(klaper::core::End e):
68         this.setName(e.name)->
69 //       this.setOutTransition(((klaper::core::Behavior)e.eContainer).transition.select(t|t.from.
       name==e.name).transformTransition())->
70         this;
71
72 private create simulator::core::Branch this transformStep(klaper::core::Branch b):
73         this.setName(b.name)->
74         this.setOutTransition(((klaper::core::Behavior)b.eContainer).transition.select(t|t.from.
             name==b.name).transformTransition())->
75         this;
76
77 private create simulator::core::Fork this transformStep(klaper::core::Fork f):
78         this.setName(f.name)->
79         this.setOutTransition(((klaper::core::Behavior)f.eContainer).transition.select(t|t.from.
             name==f.name).transformTransition())->
80         this;
81
82 private create simulator::core::Join this transformStep(klaper::core::Join j):
83         this.setName(j.name)->
84         this.setNTransition(j.transitionsNeededToGo)->
85         this.setInTransition(((klaper::core::Behavior)j.eContainer).transition.select(t|t.to.
             name==j.name).transformTransition())->
86         this.setOutTransition(((klaper::core::Behavior)j.eContainer).transition.select(t|t.from.
             name==j.name).transformTransition())->
87         this;
88
89 private create simulator::core::Wait this transformStep(klaper::core::Wait w):
90         this.setName(w.name)->
91         this.setOutTransition(((klaper::core::Behavior)w.eContainer).transition.select(t|t.from.
             name==w.name).transformTransition())->
92         this;
93
94 private create simulator::core::Activity this transformStep(klaper::core::Activity a):
```

```
95          this.setName(a.name)->
96          this.setOutTransition(((klaper::core::Behavior)a.eContainer).transition.select(t|t.from.
                name==a.name).transformTransition())->
97          this.setRepetitions(a.repetitions.getMean().toInteger())->
98          this.setServiceTime(a.internalExecTime.transformDistribution())->
99          a.internalFailTime!=null?
100                 this.setFailMode(a.internalFailTime.transformDiscreteFailMode()):this.
                        setFailMode(a.internalFailProb.transformContinuousFailMode())->
101         this;
102
103 private create simulator::core::ExternalService this transformStep(klaper::core::ServiceControl
        sc):
104         sc.binding.call!=null?
105         (
106         this.setName(sc.name)->
107         this.setOutTransition(((klaper::core::Behavior)sc.eContainer).transition.select(t|t.from
                .name==sc.name).transformTransition())->
108         sc.internalFailProb!=null?this.setFailProb(sc.internalFailProb.
                transformContinuousFailMode()):this.setFailProb(null)->
109         sc.isSynch==true?
110                 this.setServiceCall(sc.transformSynchServiceCall()):this.setServiceCall(sc.
                        transformAsynchServiceCall())->
111         this)
112         :null;
113
114 private create simulator::core::ServiceWake this transformStep(klaper::core::ServiceControl sc):
115         sc.binding.signal!=null?
116         (
117         this.setName(sc.name)->
118         this.setOutTransition(((klaper::core::Behavior)sc.eContainer).transition.select(t|t.from
                .name==sc.name).transformTransition())->
119         this.setSignal(sc.binding.signal.transformStep())->
120         this
121         )
122         :null;
123
124 private create simulator::core::DiscreteFailMode this transformDiscreteFailMode(klaper::
        probability::ProbabilityDistributionFunction di):
125         this.setInternalFailTime(di.transformDistribution())->
126         this;
127
128 private create simulator::core::ContinuousFailMode this transformContinuousFailMode(klaper::
        probability::ProbabilityDistributionFunction co):
129         this.setInternalFailProb(0.0)->
130         this;
131
132 private create simulator::core::ContinuousFailMode this transformContinuousFailMode(klaper::
        probability::Constant c):
133         this.setInternalFailProb(c.getMean())->
134         this;
135
136 private create simulator::core::ContinuousFailMode this transformContinuousFailMode(klaper::
        probability::Poisson p):
137         this.setInternalFailProb(p.getMean()) ->
138         this;
139
140 private create simulator::probability::ProbabilityDistributionFunction this
        transformDistribution(klaper::probability::ProbabilityDistributionFunction pdf):
141         this;
142
143 private create simulator::probability::Exponential this transformDistribution(klaper::
        probability::Exponential ex):
144         this.setMean(ex.getMean())->
145         this;
146
147 private create simulator::probability::Normal this transformDistribution(klaper::probability::
        Normal no):
148         this.setMean(no.getMean())->
149         this.setVariance(no.standDev.solve() * no.standDev.solve())->
150         this;
```

```
151
152 private create simulator::probability::Uniform this transformDistribution(klaper::probability::
        Uniform un):
153         this.setMin(un.min.solve())->
154         this.setMax(un.max.solve())->
155         this;
156
157 private create simulator::core::SynchServiceCall this transformSynchServiceCall(klaper::core::
        ServiceControl sc):
158         this.setService(sc.binding.call.transformService())->
159         this;
160
161 private create simulator::core::AsynchServiceCall this transformAsynchServiceCall(klaper::core::
        ServiceControl sc):
162         this.setService(sc.binding.call.transformService())->
163         this;
164
165 private create simulator::core::Transition this transformTransition(klaper::core::Transition t):
166         this.setFrom(t.from.transformStep())->
167         this.setTo(t.to.transformStep())->
168         this.setProb((t.prob>0 && t.prob<=1)?t.prob:1.0)->
169         this;
```

Appendix E

Transformation Rules from KLAPER to LQN

Here we report all the transformation rules used to go from the KLAPER meta model to the LQN meta model.

```
1  // Model transformation from the Klaper meta-model to Lqn meta-model
2  //
3  // Author: Enrico Randazzo
4  // Organization: University of Rome "Tor Vergata"
5  // email: enrico.randazzo@gmail.com
6
7
8  // imports lqn metamodel
9  import lqn;
10
11 // imports klaper metamodel packages
12 import klaper::core;
13 import klaper::probability;
14 import klaper::expr;
15
16 // and we also load the io extensions, for the purpose of
17 // debugging, in case we need it
18 extension org::openarchitectureware::util::stdlib::io;
19 extension org::openarchitectureware::util::stdlib::issues;
20 extension org::openarchitectureware::util::stdlib::counter;
21
22 // additional extensions
23 extension org::klaper::util::ProbExtensions;
24 extension org::klaper::util::ExprExtensions;
25 extension org::klaper::util::dsl::Extensions;
26
27
28 // starting point for the transformation
29 // (this extension is called from the workflow!)
30 Object klaper2lqn(klaper::core::KlaperModel m):
31 //       m.transformModel();
32          m.transformationSteps();
33
34
35 private lqn::LqnModel transformationSteps(klaper::core::KlaperModel m):
36          m.transformationStep1().transformationStep2(m);
37 //----  m.transformationStep1();
38
```

```
39
40 private lqn::LqnModel transformationStep1(klaper::core::KlaperModel m):
41         reportWarning("Model_transformation_running") ->
42         m.transformModel();
43
44
45 private lqn::LqnModel transformationStep2(lqn::LqnModel intermediate, klaper::core::KlaperModel
       m):
46         reportWarning("Model_deployment_running") ->
47         intermediate.reconfigureProcessors(m);
48
49
50 /**
51  * STEP 1 Procedures (system building without deployment):
52  * ------------------
53  */
54
55
56 /**
57  * creates the initial model with a dummy processor
58  */
59 private create lqn::LqnModel this transformModel(klaper::core::KlaperModel m):
60         this.setName("LqnModel") ->
61         this.setDescription("Model_auto-genereted_by_Klaper_tool") ->
62         this.setLqncoreSchemaVersion("1.0") ->
63         this.setLqnSchemaVersion("1.0") ->
64         this.processor.add(createDummyProcessor(m.resource)) ->          // creates a dummy
               processor
65         this.processor.addAll(m.workload.transformWorkload()) ->
66         this;
67
68
69 /**
70  * creates a dummy processor needed only to build the initial tasks infrastructure
71  */
72 private create lqn::Processor this createDummyProcessor(List resources):
73         this.setName("DummyProcessor") ->
74         this.setScheduling(lqn::SchedulingType::ps) ->
75         this.setTask(resources.typeSelect(klaper::core::Resource).transformTask()) ->
76         this.setMultiplicity(1) ->
77         this.setReplication(1) ->
78         this;
79
80
81 /**
82  * transform a workload into a processor + task
83  */
84 private create lqn::Processor this transformWorkload(klaper::core::Workload w):
85         this.setName(w.name) ->
86         this.setMultiplicity(1) ->
87         this.setReplication(1) ->
88         this.setScheduling(lqn::SchedulingType::ps) ->
89         this.setQuantum(0.1 * 0.1 * 0.1 * 0.1 * 0.1) -> // only because 0.00001 give a parse
               error! but why???
90         switch(w.type.toString())
91         {
92                 case klaper::core::WorkloadType::OPEN.toString():
93                         this.task.add(w.createOpenWorkloadTask())
94                 case klaper::core::WorkloadType::CLOSED.toString():
95                         this.task.add(w.createClosedWorkloadTask())
96                 default:
97                         reportError("For_workload_'" + w.name + "'_unknown_type_'" + w.type.
                               toString() + "'")
98         } ->
99         this;
100
101
102 /**
103  * creates a task rappresenting an open workload
104  */
```

```
105 private create lqn::Task this createOpenWorkloadTask(klaper::core::Workload w):
106         this.setName(w.name) ->
107         this.setMultiplicity(32700) ->
108         this.setReplication(1) ->
109         this.setScheduling(lqn::TaskSchedulingType::fcfs) ->
110         this.setActivityGraph(lqn::TaskOptionType::YES) ->
111         this.entry.add(w.createOpenWorkloadEntry()) ->
112         this.setTaskActivity(w.createTaskActivityGraph(w.createOpenWorkloadEntry())) ->
113         this;
114
115
116 /**
117  * creates an Entry that sets requests at an open arrival rate for an open workload
118  */
119 private create lqn::Entry this createOpenWorkloadEntry(klaper::core::Workload w):
120         this.setName(w.name.toLowerCase()) ->
121         this.setType(lqn::EntryType::NONE) ->
122         this.setOpenArrivalRate(w.arrivalProcess.getMean()) ->
123         this;
124
125
126 /**
127  * creates a task rappresenting a closed workload
128  */
129 private create lqn::Task this createClosedWorkloadTask(klaper::core::Workload w):
130         this.setName(w.name) ->
131         this.setMultiplicity(w.population) ->
132         this.setReplication(1) ->
133         this.setScheduling(lqn::TaskSchedulingType::ref) ->
134         this.setActivityGraph(lqn::TaskOptionType::YES) ->
135         this.entry.add(w.createClosedWorkloadEntry()) ->
136         this.setTaskActivity(w.createTaskActivityGraph(w.createClosedWorkloadEntry())) ->
137         this;
138
139 /**
140  * creates an Entry that sets requests for a closed workload
141  */
142 private create lqn::Entry this createClosedWorkloadEntry(klaper::core::Workload w):
143         this.setName(w.name.toLowerCase()) ->
144         this.setType(lqn::EntryType::NONE) ->
145         this;
146
147
148 /**
149  * creates an lqn Task
150  */
151 private create lqn::Task this transformTask(klaper::core::Resource r):
152         this.setName(r.name) ->
153         this.setMultiplicity(r.capacity.toInteger()) ->
154         this.setReplication(1) ->
155         this.setScheduling(r.schedulingPolicy.toLqnScheduling()) ->
156         this.setActivityGraph(lqn::TaskOptionType::YES) ->
157         this.entry.addAll(r.offeredService.createEntry()) ->
158         this.setTaskActivity(r.createTaskActivityGraph()) ->
159         this;
160
161
162 /**
163  * creates an Lqn Entry for regular tasks
164  */
165 private create lqn::Entry this createEntry(klaper::core::Service s):
166         this.setName(s.name) ->
167         this.setType(lqn::EntryType::NONE) ->
168         this;
169
170
171 /**
172  * creates an Lqn TaskActivityGraph for workload's behavior
173  */
```

```
174 private create lqn::TaskActivityGraph this createTaskActivityGraph(klaper::core::Workload w, lqn
        ::Entry entry):
175         let activities = {}:
176         let precedences = {}:
177         w.behavior.buildActivitiesGraph(activities, precedences, entry) ->
178         this.setActivity(activities) ->
179         this.setPrecedence(precedences) ->
180         this.replyEntry.add(w.behavior.step.typeSelect(klaper::core::End).first().
                createReplyEntry("reply_" + w.name)) ->
181         this;
182
183
184 /**
185  * creates an lqn TaskActivityGraph
186  */
187 private create lqn::TaskActivityGraph this createTaskActivityGraph(klaper::core::Resource r):
188         let activities = {}:
189         let precedences = {}:
190         let reply_entries = {}:
191         r.offeredService.buildActivitiesGraph(activities, precedences, reply_entries) ->
192         this.setActivity(activities) ->
193         this.setPrecedence(precedences) ->
194         this.setReplyEntry(reply_entries) ->
195         this;
196
197
198 /**
199  * converts a klaper service into a sub-graph of a TaskActivityGraph
200  */
201 private buildActivitiesGraph(klaper::core::Service s, List[ActivityDef] activities, List[
        Precedence] precedences, List[ReplyEntry] reply_entries):
202         s.behavior.buildActivitiesGraph(activities, precedences, s.createEntry()) ->
203         reply_entries.add(s.behavior.step.typeSelect(klaper::core::End).first().createReplyEntry
                ("reply_" + s.name));
204
205
206 /**
207  * creates an Lqn ReplyEntry (for TaskActivityGraph)
208  */
209 private create lqn::ReplyEntry this createReplyEntry(klaper::core::End e, String reply_name):
210         this.setName(reply_name) ->
211         this.replyActivity.add(e.createReplyActivity()) ->
212         this;
213
214
215 /**
216  * creates an Lqn ReplyActivity (for TaskActivityGraph)
217  */
218 private create lqn::ReplyActivity this createReplyActivity(klaper::core::End e):
219         this.setName(e.name) ->
220         this;
221
222
223 /**
224  * builds an activity graph (for behaviors and resources)
225  */
226 private buildActivitiesGraph(klaper::core::Behavior b, List[ActivityDef] activities, List[
        Precedence] precedences, lqn::Entry entry):
227         b.step.typeSelect(klaper::core::Start).transformStep(activities, precedences, entry) ->
228         null;
229
230
231 /**
232  * transforms a generic Step
233  */
234 private transformStep(klaper::core::Step s, List[ActivityDef] activities, List[Precedence]
        precedences, lqn::Entry entry):
235         // dummy function, do nothing
236         null;
237
```

```
238
239 /**
240  * transforms a Start step
241  */
242 private transformStep(klaper::core::Start start, List[ActivityDef] activities, List[Precedence]
        precedences, lqn::Entry entry):
243         let activity = new lqn::ActivityDef:
244         activity.setName(start.name) ->
245         activity.setHostDemandMean(0.0) ->
246         activity.setHostDemandCvsq(1.0) ->
247         activity.setThinkTime(0.0) ->
248         activity.setMaxServiceTime(0.0) ->
249         activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
250         if(entry != null) then // entry can be null for nested behaviors
251         (
252                 activity.setBoundToEntry(entry.name)
253         ) ->
254         activities.add(activity) ->
255         if(start.out.first().to.in.size == 1) then
256         (
257                 let precedence = new lqn::Precedence:
258                 precedence.setPre(start.createSingleActivityList()) ->
259                 precedence.setPost(start.out.first().to.createSingleActivityList()) ->
260                 precedences.add(precedence)
261         ) ->
262         start.out.first().to.transformStep(activities, precedences, null);
263
264
265 /**
266  * transforms an End step
267  */
268 private transformStep(klaper::core::End end, List[ActivityDef] activities, List[Precedence]
        precedences, lqn::Entry entry):
269         if(!(activities.exists(a|a.name == end.name))) then // the step is transformed only once
270         (
271                 let activity = new lqn::ActivityDef:
272                 if(end.in.size > 1) then // for (incoming) OR-join precedence
273                 (
274                         let precedence = new lqn::Precedence:
275                         precedence.setPreOR(end.createActivityListIn()) ->
276                         precedence.setPost(end.in.first().to.createSingleActivityList()) ->
277                         precedences.add(precedence)
278                 ) ->
279                 activity.setName(end.name) ->
280                 activity.setHostDemandMean(0.0) ->
281                 activity.setHostDemandCvsq(1.0) ->
282                 activity.setThinkTime(0.0) ->
283                 activity.setMaxServiceTime(0.0) ->
284                 activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
285                 activities.add(activity)
286         );
287
288
289
290 /**
291  * transforms a Wait step
292  */
293 private transformStep(klaper::core::Wait wait, List[ActivityDef] activities, List[Precedence]
        precedences, lqn::Entry entry):
294         if(!(activities.exists(a|a.name == wait.name))) then // the step is transformed only
            once
295         (
296                 let activity = new lqn::ActivityDef:
297                 if(wait.in.size > 1) then // for (incoming) OR-join precedence
298                 (
299                         let in_precedence = new lqn::Precedence:
300                         in_precedence.setPreOR(wait.createActivityListIn()) ->
301                         in_precedence.setPost(wait.in.first().to.createSingleActivityList()) ->
302                         precedences.add(in_precedence)
303                 ) ->
```

```
304                     activity.setName(wait.name) ->
305                     activity.setHostDemandMean(0.0) ->
306                     activity.setHostDemandCvsq(1.0) ->
307                     activity.setThinkTime(0.0) ->
308                     activity.setMaxServiceTime(0.0) ->
309                     activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
310                     activities.add(activity) ->
311                     if(wait.out.first().to.in.size == 1) then
312                     (
313                             let out_precedence = new lqn::Precedence:
314                             out_precedence.setPre(wait.createSingleActivityList()) ->
315                             out_precedence.setPost(wait.out.first().to.createSingleActivityList())
                                    ->
316                             precedences.add(out_precedence)
317                     ) ->
318                     wait.out.first().to.transformStep(activities, precedences, null)
319             );
320
321     /**
322      * transforms a Branch step
323      */
324     private transformStep(klaper::core::Branch branch, List[ActivityDef] activities, List[Precedence
            ] precedences, lqn::Entry entry):
325             if(!(activities.exists(a|a.name == branch.name))) then // the step is transformed only
                    once
326             (
327                     let activity = new lqn::ActivityDef:
328                     let out_precedence = new lqn::Precedence:
329                     if(branch.in.size > 1) then // for (incoming) OR-join precedence
330                     (
331                             let in_precedence = new lqn::Precedence:
332                             in_precedence.setPreOR(branch.createActivityListIn()) ->
333                             in_precedence.setPost(branch.in.first().to.createSingleActivityList())
                                    ->
334                             precedences.add(in_precedence)
335                     ) ->
336                     activity.setName(branch.name) ->
337                     activity.setHostDemandMean(0.0) ->
338                     activity.setHostDemandCvsq(1.0) ->
339                     activity.setThinkTime(0.0) ->
340                     activity.setMaxServiceTime(0.0) ->
341                     activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
342                     activities.add(activity) ->
343                     out_precedence.setPre(branch.createSingleActivityList()) ->
344                     out_precedence.setPostOR(branch.createOrList()) ->
345                     precedences.add(out_precedence) ->
346                     branch.out.to.transformStep(activities, precedences, null)
347             );
348
349     /**
350      * transforms an Activity step
351      */
352     private transformStep(klaper::core::Activity activity, List[ActivityDef] activities, List[
            Precedence] precedences, lqn::Entry entry):
353             if(!(activities.exists(e|e.name == activity.name))) then // the step is transformed only
                    once
354             (
355                     let act = new lqn::ActivityDef:
356                     let loop = new lqn::ActivityDef:
357                     let loop_name = activity.name + "_loop":
358                     let is_loop = (activity.repetitions == null) ? (false) : (!(activity.repetitions
                            .getMean() == 1.0)):
359
360                     if(activity.in.size > 1) then // for (incoming) OR-join precedence
361                     (
362                             let in_precedence = new lqn::Precedence:
363                             in_precedence.setPreOR(activity.createActivityListIn()) ->
364                             // for a single activity
365                             in_precedence.setPost(activity.in.first().to.createSingleActivityList())
                                    ->
```

```
366                 precedences.add(in_precedence)
367         ) ->
368 
369         if(is_loop || (activity.nestedBehavior != null)) then // loop condition
370         (
371                 let loop = new lqn::ActivityDef:
372                 let loop_precedence = new lqn::Precedence:
373                 let loop_list = new lqn::ActivityLoopList:
374 
375                 loop.setName(activity.name) ->
376                 loop.setHostDemandMean(0.0) ->
377                 loop.setHostDemandCvsq(1.0) ->
378                 loop.setThinkTime(0.0) ->
379                 loop.setMaxServiceTime(0.0) ->
380                 loop.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
381                 activities.add(loop) ->
382 
383                 loop_precedence.setPre((new lqn::SingleActivityList).setActivity((new
                        lqn::Activity).setName(activity.name))) ->
384                 loop_list.setEnd(activity.out.first().to.name) ->
385                 if(activity.nestedBehavior == null) then // simple activity
386                 (
387                         let loop_single_activity = new lqn::ActivityDef:
388                         let activity_loop = new lqn::ActivityLoop:
389 
390                         loop_single_activity.setName(activity.name + "_loop_item") ->
391                         loop_single_activity.setHostDemandMean(activity.internalExecTime
                                .getMean()) ->
392                         loop_single_activity.setHostDemandCvsq(1.0) ->
393                         loop_single_activity.setThinkTime(0.0) ->
394                         loop_single_activity.setMaxServiceTime(0.0) ->
395                         loop_single_activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
396                         activities.add(loop_single_activity) ->
397                         activity_loop.setName(activity.name + "_loop_item") ->
398                         activity_loop.setCount(activity.repetitions.getMean()) ->
399                         loop_list.activity.add(activity_loop)
400                 ) ->
401 
402                 if(activity.nestedBehavior != null) then // nested behavior
403                 (
404                         let activity_loop = new lqn::ActivityLoop:
405 
406                         buildActivitiesGraph(activity.nestedBehavior, activities,
                                precedences, null) ->
407                         activity_loop.setName(activity.nestedBehavior.step.typeSelect(
                                klaper::core::Start).first().name) ->
408                         (activity.repetitions != null) ? activity_loop.setCount(activity
                                .repetitions.getMean()) : activity_loop.setCount(1.0) ->
409                         loop_list.activity.add(activity_loop)
410                 ) ->
411                 loop_precedence.setPostLOOP(loop_list) ->
412                 precedences.add(loop_precedence)
413         )
414         else // single activity
415         (
416                 if(activity.nestedBehavior == null) then // simple activity
417                 (
418                         act.setName(activity.name) ->
419                         act.setHostDemandMean(activity.internalExecTime.getMean()) ->
420                         act.setHostDemandCvsq(1.0) ->
421                         act.setThinkTime(0.0) ->
422                         act.setMaxServiceTime(0.0) ->
423                         act.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
424                         activities.add(act)
425                 )
426         ) ->
427 
428         // for a single activity
429         if(activity.out.first().to.in.size == 1) then
430         (
```

```
431                         let out_precedence = new lqn::Precedence:
432                         out_precedence.setPre(activity.createSingleActivityList()) ->
433                         out_precedence.setPost(activity.out.first().to.createSingleActivityList
                                 ()) ->
434                         precedences.add(out_precedence)
435                 ) ->
436 
437                 activity.out.first().to.transformStep(activities, precedences, null)
438         );
439 
440 
441 /**
442  * transforms a ServiceControl step
443  */
444 private transformStep(klaper::core::ServiceControl service, List[ActivityDef] activities, List[
        Precedence] precedences, lqn::Entry entry):
445         if(!(activities.exists(e|e.name == service.name))) then // the step is transformed only
            once
446         (
447                 let act = new lqn::ActivityDef:
448                 let loop = new lqn::ActivityDef:
449                 let loop_name = service.name + "_loop":
450                 let is_loop = (service.repetitions == null) ? (false) : (!(service.repetitions.
                        getMean() == 1.0)):
451 
452                 if(service.in.size > 1) then // for (incoming) OR-join precedence
453                 (
454                         let in_precedence = new lqn::Precedence:
455                         in_precedence.setPreOR(service.createActivityListIn()) ->
456                         // for a single activity
457                         in_precedence.setPost(service.in.first().to.createSingleActivityList())
                                 ->
458                         precedences.add(in_precedence)
459                 ) ->
460 
461                 if(is_loop || (service.nestedBehavior != null)) then // loop condition
462                 (
463                         let loop = new lqn::ActivityDef:
464                         let loop_precedence = new lqn::Precedence:
465                         let loop_list = new lqn::ActivityLoopList:
466 
467                         loop.setName(service.name) ->
468                         loop.setHostDemandMean(0.0) ->
469                         loop.setHostDemandCvsq(1.0) ->
470                         loop.setThinkTime(0.0) ->
471                         loop.setMaxServiceTime(0.0) ->
472                         loop.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
473                         activities.add(loop) ->
474 
475                         loop_precedence.setPre((new lqn::SingleActivityList).setActivity((new
                                 lqn::Activity).setName(service.name))) ->
476                         loop_list.setEnd(service.out.first().to.name) ->
477                         if(service.nestedBehavior == null) then // simple activity
478                         (
479                                 let loop_single_activity = new lqn::ActivityDef:
480                                 let activity_loop = new lqn::ActivityLoop:
481 
482                                 loop_single_activity.setName(service.name + "_loop_item") ->
483                                 loop_single_activity.setHostDemandMean(service.internalExecTime.
                                         getMean()) ->
484                                 loop_single_activity.setHostDemandCvsq(1.0) ->
485                                 loop_single_activity.setThinkTime(0.0) ->
486                                 loop_single_activity.setMaxServiceTime(0.0) ->
487                                 loop_single_activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
488                                 activities.add(loop_single_activity) ->
489                                 activity_loop.setName(service.name + "_loop_item") ->
490                                 activity_loop.setCount(service.repetitions.getMean()) ->
491                                 loop_list.activity.add(activity_loop)
492                         ) ->
493 
```

```
494                     if(service.nestedBehavior != null) then
495                     (
496                             let activity_loop = new lqn::ActivityLoop:
497 
498                             buildActivitiesGraph(service.nestedBehavior, activities,
                                    precedences, null) ->
499                             activity_loop.setName(service.nestedBehavior.step.typeSelect(
                                    klaper::core::Start).first().name) ->
500                             (service.repetitions != null) ? activity_loop.setCount(service.
                                    repetitions.getMean()) : activity_loop.setCount(1.0) ->
501                             loop_list.activity.add(activity_loop)
502                     ) ->
503                     loop_precedence.setPostLOOP(loop_list) ->
504                     precedences.add(loop_precedence)
505             )
506             else // single activity
507             (
508                     if(service.nestedBehavior == null) then // simple activity
509                     (
510                             let call = new lqn::ActivityMakingCall:
511 
512                             // activity attributes
513                             act.setName(service.name) ->
514                             act.setHostDemandMean(service.internalExecTime.getMean()) ->
515                             act.setHostDemandCvsq(1.0) ->
516                             act.setThinkTime(0.0) ->
517                             act.setMaxServiceTime(0.0) ->
518                             act.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
519                             // service control attributes
520                             call.setFanin(1) ->
521                             call.setFanout(1) ->
522                             (service.binding.call != null) ?
523                                     call.setDest(service.binding.call.name) : call.setDest(
                                            service.binding.signal.name) ->
524                             call.setCallsMean(1.0) ->
525                             (service.isSynch) ?
526                             (
527                                     act.synchCall.add(call)
528                             )
529                             :
530                             (
531                                     act.asynchCall.add(call)
532                             ) ->
533                             activities.add(act)
534                     )
535             ) ->
536 
537             // for a single activity
538             if(service.out.first().to.in.size == 1) then
539             (
540                     let out_precedence = new lqn::Precedence:
541                     out_precedence.setPre(service.createSingleActivityList()) ->
542                     out_precedence.setPost(service.out.first().to.createSingleActivityList()
                            ) ->
543                     precedences.add(out_precedence)
544             ) ->
545 
546             service.out.first().to.transformStep(activities, precedences, null)
547     );
548 
549 
550 /**
551  * transforms a Fork step
552  */
553 private transformStep(klaper::core::Fork fork, List[ActivityDef] activities, List[Precedence]
        precedences, lqn::Entry entry):
554     if(!(activities.exists(e|e.name == fork.name))) then // the step is transformed only
            once
555     (
556             let activity = new lqn::ActivityDef:
```

```
557                     let out_precedence = new lqn::Precedence:
558                     if(fork.in.size > 1) then // for (incoming) OR-join precedence
559                     (
560                             let in_precedence = new lqn::Precedence:
561                             in_precedence.setPreOR(fork.createActivityListIn()) ->
562                             in_precedence.setPost(fork.in.first().to.createSingleActivityList()) ->
563                             precedences.add(in_precedence)
564                     ) ->
565                     activity.setName(fork.name) ->
566                     activity.setHostDemandMean(0.0) ->
567                     activity.setHostDemandCvsq(1.0) ->
568                     activity.setThinkTime(0.0) ->
569                     activity.setMaxServiceTime(0.0) ->
570                     activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
571                     activities.add(activity) ->
572                     out_precedence.setPre(fork.createSingleActivityList()) ->
573                     out_precedence.setPostAND(fork.createActivityListOut()) ->
574                     precedences.add(out_precedence) ->
575                     fork.out.to.transformStep(activities, precedences, null)
576             );
577
578
579     /**
580      * transforms a Join step
581      */
582     private transformStep(klaper::core::Join join, List[ActivityDef] activities, List[Precedence]
            precedences, lqn::Entry entry):
583             if(!(activities.exists(e|e.name == join.name))) then // the step is transformed only
                    once
584             (
585                     let activity = new lqn::ActivityDef:
586                     let in_precedence = new lqn::Precedence:
587                     in_precedence.setPreAND(join.createAndJoinList()) ->
588                     in_precedence.setPost(join.createSingleActivityList()) ->
589                     precedences.add(in_precedence) ->
590                     activity.setName(join.name) ->
591                     activity.setHostDemandMean(0.0) ->
592                     activity.setHostDemandCvsq(1.0) ->
593                     activity.setThinkTime(0.0) ->
594                     activity.setMaxServiceTime(0.0) ->
595                     activity.setCallOrder(lqn::CallOrder::STOCHASTIC) ->
596                     activities.add(activity) ->
597                     if(join.out.first().to.in.size == 1) then
598                     (
599                             let out_precedence = new lqn::Precedence:
600                             out_precedence.setPre(join.createSingleActivityList()) ->
601                             out_precedence.setPost(join.out.first().to.createSingleActivityList())
                                    ->
602                             precedences.add(out_precedence)
603                     ) ->
604                     join.out.first().to.transformStep(activities, precedences, null)
605             );
606
607     /**
608      * creates an Lqn SingleActivityList element
609      */
610     private lqn::SingleActivityList createSingleActivityList(klaper::core::Step s):
611             let activity_list = new lqn::SingleActivityList:
612             activity_list.setActivity(s.createActivity()) ->
613             activity_list;
614
615
616     /**
617      * creates an Lqn Activity
618      */
619     private lqn::Activity createActivity(klaper::core::Step s):
620             let activity = new lqn::Activity:
621             activity.setName(s.name)->
622             activity;
623
```

```
624
625  /**
626   * creates an Lqn OrList
627   */
628  private lqn::OrList createOrList(klaper::core::Branch b):
629          let or_list = new lqn::OrList:
630          or_list.activity.addAll(b.out.createActivityOr()) ->
631          or_list;
632
633  /**
634   * creates an Lqn ActivityOr
635   */
636  private lqn::ActivityOr createActivityOr(klaper::core::Transition t):
637          let activity_or = new lqn::ActivityOr:
638          activity_or.setName(t.to.name) ->
639          activity_or.setProb(t.prob) ->
640          activity_or;
641
642
643  /**
644   * creates an Lqn ActivityList
645   */
646  private create lqn::ActivityList this createActivityListIn(klaper::core::Step s):
647          this.activity.addAll(s.in.from.createActivity()) ->
648          this;
649  private create lqn::ActivityList this createActivityListOut(klaper::core::Step s):
650          this.activity.addAll(s.out.to.createActivity()) ->
651          this;
652
653  /**
654   * creates an Lqn AndJoinList
655   */
656  private create lqn::AndJoinList this createAndJoinList(klaper::core::Join join):
657          this.activity.addAll(join.in.from.createActivity()) ->
658          this.setQuorum(join.transitionsNeededToGo) -> // what is the real meaning of the quorum
                 ???? not documented!!!!
659  //      this.setQuorum(0) ->
660          this;
661
662
663  /**
664   * Step 2 procedures (system deployment):
665   * ---------------------
666   */
667
668
669  /**
670   * creates the real processors and reconfigure tasks from the dummy processor
671   */
672  private lqn::LqnModel reconfigureProcessors(lqn::LqnModel temp_model, klaper::core::KlaperModel
       m):
673  //      //---- right now hardware resources names are hardcoded but they should be defined by
       the user!!! TODO
674          temp_model.processor.addAll(m.resource.select(e|{'cpu', 'network', 'disk'}.contains(e.
                 type)).transformProcessor()) ->
675
676          // removes the dummy processor
677          temp_model.setProcessor(temp_model.processor.withoutFirst()) ->
678          temp_model;
679
680
681  /**
682   * creates an lqn processor from an hardware klaper resource
683   */
684  private create lqn::Processor this transformProcessor(klaper::core::Resource r):
685          this.setName(r.name) ->
686          this.setMultiplicity(1) ->
687  //      this.setSpeedFactor(0.0) ->
688          this.setReplication(1) ->
689  //      this.setQuantum(0.1) ->
```

```
690         (r.schedulingPolicy != null) ? this.setScheduling(r.schedulingPolicy.transformScheduling
                ()) : null ->
691 //      this.setScheduling(lqn::SchedulingType::ps) ->
692         this.setQuantum(0.1 * 0.1 * 0.1 * 0.1 * 0.1) -> // only because 0.00001 give a parse
                error! but why???
693
694         // adds the main task running on this processor (the task that represents hardware
                services)
695         this.task.add(r.transformTask()) -> // it uses caching!!!!
696
697         // looks for all the tasks that use the previous one
698         this.task.addAll(
699                 ((klaper::core::KlaperModel)(r.eContainer)).resource.select(e | e != r).select(e
                        | e.offeredService.behavior.step.typeSelect(klaper::core::ServiceControl).
                        binding.call.name.intersect(r.transformTask().entry.name).size > 0).
                        transformTask()
700         ) ->
701         this;
702
703 private lqn::SchedulingType transformScheduling(klaper::core::SchedulingPolicyKind s):
704         // this switch seems it doesn't work with normal enum values. With strings it works...
                but why not with enum literals?????
705         switch(s.toString()) {
706                 case klaper::core::SchedulingPolicyKind::EarliestDeadlineFirst.toString(): lqn::
                        SchedulingType::hol
707                 case klaper::core::SchedulingPolicyKind::FIFO.toString(): lqn::SchedulingType::
                        fcfs
708                 case klaper::core::SchedulingPolicyKind::FixedPriority.toString(): lqn::
                        SchedulingType::pp
709                 case klaper::core::SchedulingPolicyKind::LeastLaxityFirst.toString(): lqn::
                        SchedulingType::hol
710                 case klaper::core::SchedulingPolicyKind::RoundRobin.toString(): lqn::
                        SchedulingType::ps
711                 case klaper::core::SchedulingPolicyKind::TimeTableDriven.toString(): lqn::
                        SchedulingType::ps
712                 default: lqn::SchedulingType::NULL
713         };
714
715 private lqn::TaskSchedulingType toLqnScheduling(klaper::core::SchedulingPolicyKind s):
716         switch(s.toString()) {
717                 case klaper::core::SchedulingPolicyKind::EarliestDeadlineFirst.toString(): lqn::
                        TaskSchedulingType::hol
718                 case klaper::core::SchedulingPolicyKind::FIFO.toString(): lqn::
                        TaskSchedulingType::fcfs
719                 case klaper::core::SchedulingPolicyKind::FixedPriority.toString(): lqn::
                        TaskSchedulingType::pri
720                 case klaper::core::SchedulingPolicyKind::LeastLaxityFirst.toString(): lqn::
                        TaskSchedulingType::hol
721                 case klaper::core::SchedulingPolicyKind::RoundRobin.toString(): lqn::
                        TaskSchedulingType::poll
722                 case klaper::core::SchedulingPolicyKind::TimeTableDriven.toString(): lqn::
                        TaskSchedulingType::poll
723                 default: lqn::TaskSchedulingType::NULL
724
725         };
```

Bibliography

[1] Seventh european framework programme. http://ec.europa.eu/research/fp7/.

[2] Algirdas Avizienis, Jean Cloude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, pages 11–33. IEEE, 2004.

[3] Egon Boerger. High level system design and analysis using abstract state machines.

[4] Egon Boerger and Robert F.Stark. *Abstract State Machines. a Method for High-Level System Design and Analysis.* Springer, 2003.

[5] Gunter Bolch, Stefan Greiner, Herman de Meer, and Kishor Shridharbhai Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Application.* Wiley-Interscience, 2006.

[6] E. Borger, A. Cavarra, and E. Riccobene. An asm semantics for uml activity diagram.

[7] Egon Borger. Asm tutorial. http://www.di.unipi.it/boerger/asmtutorialetaps.html.

[8] Federico Carbonetti. Tecniche model-driven per l'analisi automatizzata dell'affidabilità di sistemi a componenti, 2009. Università degli studi di Roma "Tor Vergata".

[9] Fabrizio D'Ammassa. Analisi predittiva di requisiti software non funzionali tramite simulazione ad eventi discreti: un approccio model driven, 2008. Università degli studi di Roma "Tor Vergata".

[10] Eclipse. http://www.eclipse.org.

[11] Eclipse. Acceleo. http://www.acceleo.org/pages/home/en.

[12] Eclipse. Atl. atlas transformation language. http://www.eclipse.org/m2m/atl/.

[13] Eclipse. Graphical modeling framework. http://www.eclipse.org/modeling/gmf/.

[14] Eclipse. Jet. http://www.eclipse.org/modeling/m2t/?project=jet#jet.

[15] Eclipse. Operational qvt language (qvto). http://wiki.eclipse.org/M2M/Operational_QVT_Language_(QVTO).

[16] Eclipse. Qvt declarative (qvtd). http://wiki.eclipse.org/M2M/Relational_QVT_Language_(QVTR).

[17] Eclipse. Xpand. http://wiki.eclipse.org/Xpand.

[18] Eclipse. Xtend. http://wiki.eclipse.org/Xpand.

[19] Betty H.C.Chen et al. *Software Engineering for Self-Adaptive systems*, chapter Software Engineering for Self-Adaptive Systems: A Research Rodmap. LNCS, 2009.

[20] Kent Beck et al. http://agilemanifesto.org/.

[21] William Feller. *An Introduction to Probability Theory and its Applications, Vol. 1*. Wiley, 1968.

[22] Greg Franks, Peter Maly, Murray Woodside, Dorina C.Petriu, and Alex Hubbard. *Layered Queueing Network Solver and Simulator User Manual*. Department of Systems and Computer Engineering Carleton University, 2005.

[23] Greg Franks, Peter Maly, Murray Woodside, Dorina C.Petriu, and Alex Hubbard. *Layered Queueing Network Solver and Simulator User Manual*. Department of Systems and Computer Engineering of Caleton University, revision 6840 edition, 12 2005. site_address.

[24] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages.

[25] K. Goseva-Popstojanova, A.P. Matur, and K.S. Trivedi. Architecture-based approach to reliability assessment of software systems, 2001.

[26] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. *WOSP*, 2007.

[27] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, 1997.

[28] Yuri Gurevich. *Specification and validation methods*. Oxford University Press, 1995.

[29] IEEE. *IEEE 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*.

[30] M.R. Lyu. Handbook of software reliability engineering. IEEE Computer Society Press, 1996.

[31] Peter Mali. Description of the lqn xml schema.

http://www.sce.carleton.ca/rads/lqns/lqn-documentation/schema/.

[32] Eclipse Modeling. http://www.eclipse.org/home/categories/index.php?category=modeling.

[33] University of Edimburg. Simjava homepage. http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/.

[34] Karlsruhe Institute of Technology. Palladio component model. http://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model.

[35] OMG. Mda guide. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[36] OMG. Mda specification. http://www.omg.org/mda/specs.htm.

[37] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. http://www.omg.org/spec/QVT/1.0/.

[38] OMG. Mof specification. http://www.omg.org/spec/MOF/2.0/.

[39] OMG. Object constraint language. http://www.omg.org/spec/OCL/2.0/.

[40] OMG. *OMG Systems Modeling Language (OMG SysML^{TM})*, version 1.1 edition.

[41] OMG. Uml 2.2 superstructure and infrastructure. http://www.omg.org/spec/MOF/2.0/.

[42] OMG. Uml profile for modeling and analysis of real-time and embedded systems (marte). http://www.omg.org/spec/MARTE/1.0/.

[43] OMG. Uml profile for schedulbility, performance and time, version 1.1. http://www.omg.org/cgi-bin/doc?formal/2005-01-02.

[44] OMG. Xml metadata interchange. http://www.omg.org/spec/XMI/2.1.1/.

[45] openArchitectureWare. *Xtend Reference*. http://www.openarchitectureware.org/pub/docum

[46] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise system. Journal of Object technology, 2008.

[47] Klaper project. http://valerianus.ce.uniroma2.it/wiki/klaper:start.

[48] Puma project. http://www.sce.carleton.ca/rads/puma/.

[49] Q-Impress project. http://www.q-impress.eu/wordpress/.

[50] Balsamo Simonetta, DiMarco Antinisca, Inverardi Paola, and Simeoni Marta. Model-based performance prediction in software development: A survey. In *IEEE transactions on software engineering*.

[51] C.U. Smith. *Performance Engineering of Software systems*. Addison Wesley, 1990.

[52] Artisan Studio. http://www.artisansoftwaretools.com/products/artisan-studio/.

[53] Kishor S. Trivedi and Robin Sahner. Sharpe at the age of twenty two.

[54] University of Edimburg. *The SimJava Tutorial*, 2002. http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/guide/tutorial.html.

[55] Gerald Weinberg. Quality software management: Systems thinking.

[56] Wikipedia. Markov chain. http://en.wikipedia.org/wiki/Markov_chain.

[57] Wikipedia. Markov process. http://en.wikipedia.org/wiki/Markov_process.

[58] Wikipedia. Performance engineering. http://en.wikipedia.org/wiki/Performance_Engineering.

[59] Wikipedia. Probability mass function. http://en.wikipedia.org/wiki/Probability_mass_function.

[60] Wikipedia. Qvt. http://en.wikipedia.org/wiki/QVT.

[61] Wikipedia. Simulation. http://en.wikipedia.org/wiki/Simulation.

[62] Wikipedia. Software development methodologies. http://en.wikipedia.org/wiki/Software_development_methodology.

[63] Wikipedia. Software quality. http://en.wikipedia.org/wiki/Software_quality.

[64] J. Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP*, pages 1–12. ACM, 2008.