On the Model-Driven Performance and Reliability Analysis of Dynamic Systems

Thesis written by
Javier Troya Castilla
and supervised by
Prof. Antonio Vallecillo Moreno

December, 2012

El Dr. Don Antonio Vallecillo Moreno, Catedrático de Universidad del Área de Lenguajes y Sistemas Informáticos de la E.T.S. de Ingeniería Informática de la Universidad de Málaga,

Certifica que D. Javier Troya Castilla, Ingeniero Informático, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo mi dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

On the Model-Driven Performance and Reliability Analysis of Dynamic Systems

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, diciembre de 2012

Fdo.: Antonio Vallecillo Moreno
Catedrático de Universidad
Área de Lenguajes y Sistemas Informáticos

Contents

List of Figures

List of Tables

1 Introduction

When designing and developing any kind of system, the first thing engineers think about is its functionality. They try to identify the actions or functions that the system must be able to perform, by identifying its functional requirements. However, it is crucial in many systems not to forget about the non-functional requirements, which qualify *how* the system must behave. Non-functional requirements are constraints upon the systems behavior.

If we think, for example, of systems supported by software, the most critical the software is, the most important the compliance with its non-functional requirements becomes. There are known cases where the non-compliance of these requirements resulted in the loss of money, material things, or even people. The dissertation by Rodríguez-Dapena [RD02] reports some of these cases of failures. To mention some of them, 130 injuries were reported in 1998 due to unjustified and sudden deployment of automobile airbags. Almost one million cars where recalled by General Motors due to this failure. Some cars had a sensor calibration problem which made the airbags inflate under normal conditions on paved roads. The fix involved a little software reprogramming. A problem in the software of traffic lights occurred in 1996 in Washington D.C. annoyed the people who had to be on time to work. Most traffic lights in downtown went onto their weekend schedule (15 seconds of green per light), rather than their rush hour schedule (50 seconds of green per light). This caused traffic jams during the morning rush hour. The problem was caused by a new version of the software installed in the central system that controls all the traffic lights. A software problem also made the Mars Polar Lander crash on Mars' surface in 1999. The software was responsible for turning on the engines that would make the space probe slow down, but they did not turn on, so the probe reached the surface at a high speed and crashed. All these failures could have been avoided if the non-functional requirements had been properly defined, designed, implemented and tested.

Knowing that non-functional requirements have to be taken into account, another issue to consider is *when* and *how* they should be specified, implemented and tested. Many approaches fail to leverage the testing of the non-functional requirements to the latest stages of the software life cycle or in the construction of any kind of system. In Figure 1.1, we can see some approaches for testing non-functional requirements. Approaches a) and b) follow practices we consider should

not be followed, while approach c) is the ideal one. The example of the traffic lights described before follows the approach a). Thus, the failure was not detected until the new software was actually deployed in the central system that controls all the traffic lights. The same happened in the other two examples presented. In other systems, non-desirable reverse engineering is applied. Ideally, the design defines what is to be implemented. However, in real-life projects, this is not always the case. In fact, during the implementation phase, engineers often realize that the design needs to be modified, so they first modify the implementation and then propagate the changes towards the design—approach b). It also happens very often that projects run out of time due to strict deadlines. In these cases, the implementation phase forgets completely about the design phase, and the changes addressed in the implementation are not even propagated back to the design. This is specially detrimental for the long-term system maintenance, since the design and implementation will no longer concord.

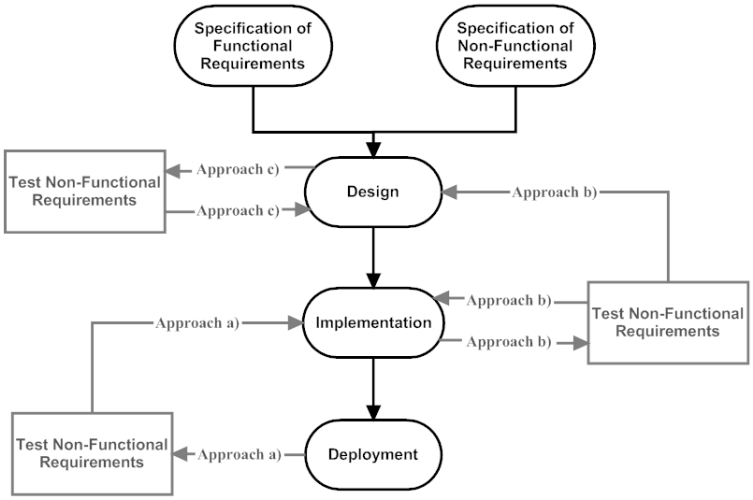


Figure 1.1: Non-functional requirements in the software life cycle.

In software systems, the design, implementation and deployment phases are all realized in the same medium: the computer. Reversing changes from deployment to implementation (approach a in Fig. 1.1) or from implementation to design (approach b) is detrimental in this kind of systems. However, at least, the medium does not need to be changed when reversing the changes. The situation is more critical in the case of non software systems, or where the software represents a small part of the whole system, like in embedded systems. Thus, the design and a prototypical implementation can be realized in a computer, but the deployment needs to be set in a real space and with real material. Reversing changes in these cases is going to be surely expensive, since material components may need to be replaced or even rejected after deployed. The three examples presented before correspond to this kind of systems.

1.1 Approach and Methodology

One of our goals is to be able to test non-functional requirements before the implementation (approach c in Figure 1.1) and, consequently, independently of the platform. We make use of the Model-Driven Engineering (MDE) methodology. MDE was born as a promising approach to address platform complexity and overcome the inability of third-generation languages to alleviate this complexity and express domain concepts effectively. The basis of MDE is the use of models as first class entities throughout the software engineering life cycle. According to [KWB03], "A model is a description of (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer". Consequently, in MDE models are not to be used as mere documentation, but they are key artifacts from which the whole system can be derived and implemented.

MDE technologies apply lessons learned from earlier efforts and offer several mechanisms to develop higher-level platform and language abstractions. In this way, instead of general-purpose notations and languages, such as the well-known languages Java or C, which rarely express application domain concepts and design intent, domain-specific languages (DSLs) can be created via metamodeling to precisely match the domain's semantics and syntax, so they are closer to the problem domain than to the implementation domain. Having DSLs allows to relate graphic elements directly to a familiar domain. It gives rise to the so-called domain-specific visual languages (DSVLs). They do not only help reducing the learning process, but also help a broader range of subject matter experts, such as system designers and software architects, ensure that software systems meet user needs. Another crucial MDE technology is model transformations, whose application has become increasingly commonplace in MDE, with a number of stable model transformation languages and tools available.

An important aspect of MDE tools, crucial for the purpose we pursue, is their imposition of domain-specific constraints and performance of model checking, which can detect and prevent many errors early in the life cycle. Furthermore, MDE tool generators do not need to be as complicated as those in the 1980s and 1990s, since they can synthesize artifacts that map onto higher-level middleware platform APIs and frameworks, rather than lower-level OS APIs. Consequently, it is often much easier and more lightweight to develop, debug, and evolve MDE tools and applications created with these tools. MDE already leverages, augments and integrates several technologies such as patterns, model checkers, third-generation and aspect oriented languages, application frameworks, component middleware platforms, product-line architectures, etc.

There already exist several proposals for modeling the structure and functional behavior of systems in the earlier stages of the software development, but do not deal with non-functional requirements. Some of these proposals also come with

supporting environments for animating or executing the specifications, based on the transformations of the models into other models that can be executed [dLV06, dLV08, EHC05, EE08, EHKZ05a].

As already mentioned, the correct and complete specification of a system includes other aspects beyond structure and basic behavior. In particular, the specification and analysis of its non-functional requirements is critical in many important distributed application domains including embedded systems, multimedia applications, and e-commerce services and applications. In the last few years, researches have faced the challenge of defining quantitative models for non-functional specification and validation from software artifacts [BMIS04, Zsc10]. Several methodologies have been introduced, all sharing the idea of annotating software models with data related to non-functional aspects, and then translating the annotated models into models ready to be validated [CMI07, FBV+09, FJA+09]. Most of these proposals for annotating models with non-functional requirements exist for UML-based notations, with UML Profiles such as UML-QoSFT, UML-SPT or MARTE [OMG05, OMG04, OMG08]. Several tools already exist for analyzing and simulating such models and for carrying out performance analyses.

Although these profiles provide solutions for UML models, the situation is different when DSVLs are used to specify a system. In the first place, non-functional requirements are normally written using languages which are completely alien to system designers, because such languages are typically influenced by the analysis methods that need to be used, and written in the languages of these methods. Besides, their level of abstraction is normally lower than the DSVL notations used to specify the system, and tend to be closer to the solution domain than to the problem domain (in contrast with the problem-domain orientation of DSVLs). Furthermore, when several properties need to be analyzed, the annotated models become cluttered with a plethora of different annotations and marks (see many of the diagrams shown in the MARTE specification [OMG08]). Another problem is that current proposals for the specification of these kinds of properties tend to require skilled knowledge of specialized languages and notations, which clashes with the intuitive nature of end-user DSVLs and hinders its smooth combination with them. Finally, most of these proposals specify non-functional requirements and constraints using a *prescriptive* approach, i.e., they annotate the models with a set of requirements on the behavior of the system and with constraints on some model elements, but are not very expressive for describing how such values are dynamically computed or evolve over time.

1.2 Contribution

In this dissertation we present an alternative approach to specify the properties that need to be analyzed, integrating new objects in the specifications that allow them to be captured in the early stages of the software development—approach c) in Figure 1.1. Among all the available non-functional requirements, such as

safety, security, usability, reliability and performance, we focus in the last two. Our proposal is based on the *observation* of the execution of the system actions and of the state of its constituent objects. We use this approach to simulation and analysis in the case of DSVLs that specify behavior in terms of rules describing the evolution of the modeled artifacts along some time model. We show how, given an initial specification of the system, the use of *observer* objects enables the analysis of non-functional requirements in early phases of the software life cycle. The key of these observers is that they can be specified in the same language that the domain expert is using for describing the system, so no further notations need to be done. Thus, the idea is to merge together the specification of systems and observers in order to combine them from the design phase (Figure 1.2), considering that non-functional requirements are specified by these observers.

We do not only consider software systems, but also those that are to be deployed in a medium different than the computer. Concretely, we focus on *dynamic* systems. By "dynamic" systems we mean systems where there are *things* flowing in. These "things" can be data, components, part of components, etc. By "flowing" we understand that these things evolve within the system, by for example changing among different states or going from one component to another. We will present several use cases of this kind of systems throughout this dissertation. For example, a *production line system* for producing hammers where hammer's parts move from one machine to another is a dynamic system. Another example is the *check-in process* in an airport, where the passengers go through different states: waiting in queue, check-in realized, waiting for their plain, etc. Any network is also considered a dynamic system, since packets flow among servers. We also deal with a formalism with which dynamic systems can be modeled: queuing networks.

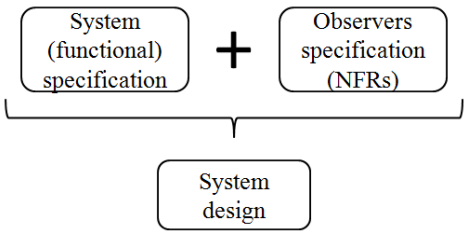


Figure 1.2: Integrating observers from the specification.

A tool presented in a former thesis [Riv10] has served as a supporting environment for the proposal. It is called *e-Motions* and it is a tool for the specification of real-time systems [RDV09, RVD09, RDV10]. Several extensions to such tool are presented in this dissertation. We also describe a methodology for the specification of non-functional properties in rule-based DSVLs, and present a modular approach to decouple the specification of observers with regards to the functional specification of systems.

In this thesis we also deal with the challenge of the MDE community to build semantic bridges between modeling spaces [DGFD06], so that programs or models can travel from one to the other, being able to use the local analysis tools to reason about their properties, conduct simulations, validations, etc. [Val08]. These bridges can be naturally specified in the context of MDE by means of model transformations. Bridging modeling domains is easier when they have similar expressiveness, because in this case model transformations can be naturally defined between them. However, the situation is not so easy when the expressiveness of the two domains to relate is rather disparate. In our case, we were originally interested in bridging our notation and associated toolkit for real-time systems modeling, *e-Motions*, to other notations and tools (Figure 1.3). The goal was to be able to conduct different kinds of analysis on the models, beyond the ones allowed by *e-Motions*. One of the first choices was Queueing Network Models (QNMs), given its possibilities for conducting performance analysis of the models in an analytical way [DB78]. However, we soon discovered that most of the information available in the *e-Motions* models was lost or severely demeaned when translated into QNMs. For example, the (rather complex) OCL expressions with the algorithms for selecting the next elements to move in some of our systems were reduced to just probabilities, which had to be estimated somehow (normally using simulations, which precisely were the analysis we tried to avoid by the use of analytical methods). Similarly, determining the *e-Motions* model elements that should be transformed into servers, jobs or queues was not a trivial task, and even not possible in some situations without human intervention.

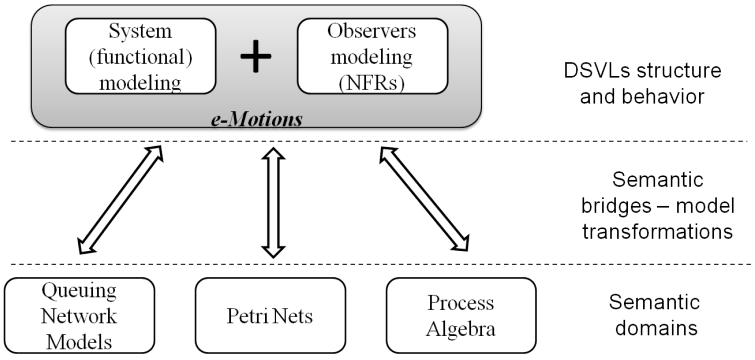


Figure 1.3: Semantic bridges.

We soon discovered, however, that the reverse mapping was not only feasible, but even easy to implement. Building a bridge between QNMs and *e-Motions* would allow using the *e-Motions* simulation and analysis tools with QN models, bringing along several interesting benefits. For instance, we could easily support arbitrary probabilistic distributions for arrival and service times, beyond the usual Exponential and Uniform distributions; we could also leverage the advantages of MDE for building, in a high-level and modular way, a domain-specific language

(DSL) for modeling QNMs and a supporting toolkit for simulating them; we could use (an Ecore version of) PMIF [Smi10], a performance model interchange format, for defining and exchanging QN models between tools; finally, the bridge would also allow to define a behavioral semantics for QNMs, specified by means of a fixed set of behavioral rules in *e-Motions*. Regarding the last point and considering the possibilities and expressive power for modeling DSLs in *e-Motions*, such behavioral semantics can be decorated with new features in order to extend the QNMs' behavior in a more realistic manner. Concretely, we have extended this behavior by considering that servers can fail, and can be temporarily unavailable or out of order. This dissertation reports on the experience on building that bridge, and describes the DSL and the tool (called xQNM) that we have developed using standard MDE techniques. Moreover, we show how xQNM is comparable to other existing QNM tools which were defined using standard programming approaches, despite being defined at a higher-level of abstraction. We shall present the xQNM architecture, based on a layered approach, and the implementation details that contribute to achieve the above mentioned advantages.

As a matter of notation, we will be using throughout this document the terms non-functional requirements (NFRs), non-functional properties (NFPs) and quality of service (QoS) properties indistinctly.

1.3 Publications

This section is dedicated to present the publications in journals and conferences, with peer-review, that support this dissertation. Additionally, it enumerates those publications, also with peer-review, where the author of this dissertation is involved but which are not related to the work presented here (although most of them are within the same domain).

1.3.1 Publications Supporting this Dissertation

International Journals

- Javier Troya, Antonio Vallecillo, Francisco Durán and Steffen Zschaler. Model-driven performance analysis of rule-based domain specific visual models. *Information and Software Technology*, 55(1): 88–110. January 2013. This paper supports most of the content presented in Chapter 3 and part of the approach presented in Chapter 6.

- Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10(5): 1–29, 2011. The work presented in this paper is not a main contribution in this dissertation, although it is referenced in a small example shown in Section 2.1.2.

International Journals under review

- Javier Troya and Antonio Vallecillo. Specification and Simulation of Queuing Network Models using Domain-Specific Languages. *Computer Standards & Interfaces*. Submitted. This paper presents the contributions of Chapter 5.

International Conferences

- Francisco Durán, Steffen Zschaler and Javier Troya. On the Reusable Specification of Non-functional Properties in DSLs. To appear in *Proceedings of the 5th International Conference on Software Language Engineering (SLE 2012)*. Dresde, Germany, September 25th - 28th, 2012. This paper presents part of the approach shown in Chapter 6.

- Javier Troya, José Bautista, Fernando López-Romero and Antonio Vallecillo. Lightweight Testing of Communication Networks with *e-Motions*. In *Proceedings of the 5th Internation Conference on Tests & Proofs (TAP 2011)*, LNCS 6706, 187–204, Springer. Zurich, Switzerland. June 30th - July 1st, 2011. The case study presented in this paper is explained in Section 3.6.

- Javier Troya and Antonio Vallecillo. Towards a Rewriting Logic Semantics for ATL. In *Proceedings of the Third International Conference on Model Transformations (ICMT 2010)*, LNCS 6142, 230–244, Springer. Málaga, Spain, June 28th - 29th, 2010 (**Best Paper Award**). This paper is the conference version of the paper "A Rewriting Logic Semantics for ATL". Although the work presenter in this paper is not a main contribution in this dissertation, it is referenced in a small example shown in Section 2.1.2.

- Javier Troya, José E. Rivera and Antonio Vallecillo. Simulating Domain Specific Visual Models by Observations. In *Proceedings of the Symposium on Theory of Modeling and Simulation (DEVS 2010)*. Orlando, Florida, USA. April 11th - 15th, 2010. Part of the work presented in this paper is contained in Chapter 3.

International Workshops

- Javier Troya and Antonio Vallecillo. A Domain Specific Visual Language for Modeling Power-Aware Reliability in Wireless Sensor Networks. To appear in *Proceedings of the Fourth International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NFPinDSML 2012), MODELS 2012 Workshops*. Innsbruck, Austria, Sept. 30th - Oct. 2nd, 2012. This paper presents most of the work in Chapter 4.

- Javier Troya, José E. Rivera and Antonio Vallecillo. On the Specification of Non-Functional Properties by Observation. In *Proceedings of the Second International Workshop on Non-functional Properties in Domain Specific Modeling Languages (NFPinDSML 2009), MODELS 2009 Workshops*. LNCS 296–309, Springer. Denver, Colorado, USA, October 4th, 2009. This paper means the starting point for all the contributions presented in this dissertation. It was favorably received by the audience and that encouraged us to continue working on the (at that point) naive approach presented in it.

National Conferences

- Javier Troya, Antonio Vallecillo and Francisco Durán. On the Modular Specification of Non-Functional Properties in DSVLs. *XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2012)*. Almería, Spain. September 17th - 19th, 2012. This paper represents a starting point for the approach presented in Chapter 6.

- José M. Bautista, Javier Troya and Antonio Vallecillo. Diseño y Monitorización de Sistemas de Colas con *e-Motions*. *XVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2011)*. A Coruña, Spain. September 5th - 7th, 2011 (**Best Paper Award**). The example of the check-in process in an airport shown in Chapters 2 and 6 is a modification of the case study presented in this paper.

Technical Reports (Non-Published)

- Javier Troya, José M. Bautista and Antonio Vallecillo. A Rewriting Logic Semantics for ATL (Extended Version). Technical Report of the paper "A Rewriting Logic Semantics for ATL". 2011.

- Javier Troya and Antonio Vallecillo. Towards a Rewriting Logic Semantics for ATL (Extended Version). Technical Report of the paper "Towards a Rewriting Logic Semantics for ATL". 2010.

1.3.2 Further Publications

International Journals under review

- Shekoufeh Kolahdouz-Rahimi, Kevin Lano, Suresh Pillay, Javier Troya and Pieter Van Gorp. Goal-oriented Measurement of Model Transformation Methods. *Science of Computer Programming*. Submitted.

International Conferences

- Moisés Castelo Branco, Javier Troya, Krzysztof Czarnecki, Jochen Küster and Hagen Völzer. Matching Business Process Workflows Across Abstrac-

tion Levels. In *Proceedings of the ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012)*. Springer Verlag, LNCS, 626–641. ISBN: 978-3-642-33665-2. Innsbruck, Austria, Sept. 30th - Oct. 5th, 2012.

- Javier Criado, Luis Iribarne, Nicolás Padilla, Javier Troya and Antonio Vallecillo. An MDE approach for Runtime Monitoring and Adapting Component-based Systems: Application to WIMP User Interfaces Architectures. In *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2012)*. IEEE Computer Society, 150–157. ISBN: 978-1-4673-2451-9. Cesme, Izmir, Turkey, Sept. 5th - 8th, 2012.

- Manuel Díaz, Daniel Garrido and Javier Troya. Developing a Communications Architecture Based on WCF for Use in Nuclear Power Plant Simulators. In *Proceedings of the IADIS International Conference, Applied Computing 2009*. IADIS Press, 171–175. ISBN: 978-972-8924-97-3. Rome, Italy, Nov. 19th - 21st, 2009.

National Conferences

- Juan F. Inglés-Romero, Cristina Vicente-Chicote, Javier Troya and Antonio Vallecillo. Prototyping Component-Based Self-Adaptive Systems with Maude. *XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2012)*. Almería, Spain. September 17th - 19th, 2012.

- Javier Criado, Luis Iribarne, Nicolás Padilla, Javier Troya and Antonio Vallecillo. Adapting Component-based User Interfaces at Runtime using Observers. *XVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2011)*. A Coruña, Spain. September 5th - 7th, 2011.

1.4 Outline

The remaining chapters in this dissertation are structured as follows:

Chapter 2: Background. It presents some concepts, technologies and tools that have served as basis for this dissertation. The Model-Driven Engineering methodology is presented, along with its most relevant features, such as models, metamodels, domain-specific languages or model transformations. The *e-Motions* tool is also presented and a case study is shown, since many of the approaches presented throughout this document have been implemented in this tool.

Chapter 3: Model-Driven Performance Analysis of Rule-Based DSVLs. It presents the *observers* model for achieving a high-level analysis of non-functional

properties. This chapter focuses on performance. A general methodology for the modeling and analysis of non-functional properties is presented, and it is applied to get the performance measures of two different systems. This chapter also presents an extension to *e-Motions*: the integration of probability distributions, key in the modeling of realistic real-time systems.

Chapter 4: A DSVL for Modeling Power-Aware Reliability in WSNs. This chapter focuses on reliability. It explains how the addition of probabilistic rules to *e-Motions* allows the modeling of state machines of systems' components, and how the probabilistic distributions implemented in the previous chapter help modeling these kinds of systems. The chapter also describes how our approach is appropriate for analyzing the reliability of a protocol for wireless sensor networks, and how such protocol can be easily modified with our approach in order to test new variants. The chapter ends with the analysis of the results gathered through simulation.

Chapter 5: Specification and Simulation of QNMs using DSLs. It reports on the experience on building a semantic bridge from queuing network models to *e-Motions*. The result is a domain-specific language integrated in a tool, built on top of *e-Motions*, which allows to graphically depict QNMs, import and export them, and simulate them to obtain performance measures.

Chapter 6: A Modular Approach for the Specification of Observers. It presents an alternative approach to that in Chapter 3 for including observers in the specification of systems. We propose the definition of observer languages independently from the definition of systems. Then, observer languages are to be woven with system specifications in order to include observers in the latter.

Chapter 7: Related Work. This chapter surveys and describes works related to the contributions presented throughout this dissertation. It surveys works in the area of high-level modeling and high-level analysis and monitoring of non-functional properties, works aimed at building the semantic gap between different semantic domains, and works in the general area of modular definition of languages, models and transformations.

Chapter 8: Conclusions and Future Work. It summarizes the contributions presented in the different chapters and outlines possible future work.

Appendix A: Probabilistic Distributions in Maude. This appendix presents the Maude implementation of several probabilistic distributions, explained in Chapter 3, integrated in *e-Motions*.

Appendix B: ATL Transformations for weaving DSLs. It shows the implementation, in the ATL Transformation Language, of the approach presented in Chapter 6.

Appendix C: Metamodels Used Throughout this Dissertation. It shows and describes the main metamodels we have created, studied and used in this dissertation.

Appendix D: Resumen. In this appendix we summarize this dissertation, in Spanish.

Appendix E: Conclusiones y Contribuciones. In this appendix we list the contributions of this dissertation, in Spanish.

2 Background

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a methodology that advocates the use of models as first class entities throughout the software engineering life cycle. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design and promoting communication between individuals and teams working on the system.

2.1.1 A Bit of History

The concepts of models and modeling are not new. Since at least four decades, software engineers have been creating abstractions that would help them program in terms of their design intent rather than being constrained by the underlying computing environment. The key was to raise the level of abstraction, what would abstract them away from the complexities of the programming environment. As an example, third generation languages, such as C (born in the early 1970's), raised the level of abstraction over assembly languages. In this way, programmers did not need to worry about which area of the machine's memory they would access. In a same manner, early operating system platforms, such as OS/360 (released in 1967) and Unix (originally developed in 1969), shielded developers from the complexities of programming directly with hardware devices [Sch06].

Later, the appearance of computer-aided software engineering (CASE) in the 1980's aimed at providing a graphical means of simplifying software development, whilst also generating implementation artifacts. However, according to [GS03], they were based on proprietary modeling languages and ineffective code generators. During the past two decades, the advances in programming languages and platforms have raised the level of software abstractions available to developers. Examples of this are the object-oriented languages like C++, Java, or C#, which offer a higher level of abstraction than Fortran or C. However, they still had a distinct computing-oriented focus. This means that they provided abstractions of the solution space (i.e., the domain of computing technologies themselves) rather than abstractions of the problem space that express designs in terms of concepts in application domains, such as aerospace, biology and telecom.

Apart from the already commented challenges, some other are still to be faced. One of the most important issues is the rapidly growing complexity of systems, moving faster than software development technologies can cope with. Furthermore, the fast evolution of platforms and the appearance of new ones make developers spend a great amount of effort on porting legacy applications. How to deal with all this? MDE tries to offer a solution. It reuses the idea of compilers, and model transformations tools represent the core of the paradigm [MSU+04] since they free developers from having to deal with low level elements.

2.1.2 Main Concepts

Models

The fundamental concept in model-driven approaches is that of models. In short, we can define a model as a simplified abstraction of a system or real world concept. However, this is not the only definition that can be found in the literature. For Warmer and Kleppe [WK03], a model is a description of a system, or part of it, written in a well defined language. According to Seidewitz [Sei03], a model is a set of statements about some system under study. Here, *statement* means some expression about the system that can be considered true or false. The OMG gives different definitions for model in different documents. Thus, in [OMG01] it is defined as the representation of a part of the functionality, structure and/or behavior of a system. In [OMG03], the OMG defines a model as the description or specification of a system and its environment defined for a concrete purpose. Finally, in [OMG10] the OMG states that a model captures a view of a physical system, with a concrete purpose. The purpose determines what is to be included in the model and what is irrelevant. Consequently, the model describes those aspects of the physical system which are relevant for the model's purpose, and at the right level of abstraction.

Use and utility of models. There is no consensus on what are the features that models should have in order to be useful and effective. However, one of the most accurate descriptions is that of Bran Selic, one of the founders of MDE and a pioneer in the use of its techniques. In a presentation that he gave in the MODPROD congress in Sweden in 2011 entitled *Abstraction Patterns in Model-Based Engineering*, he described which characteristics useful engineering models should have. They should be **purposeful**, constructed to address a specific set of concerns/audience; **abstract**, emphasizing important aspects while removing irrelevant ones; **understandable**, expressed in a form that is readily understood by observers; **accurate**, by faithfully representing the modeled system; **predictive**, so that they can be used to answer questions about the modeled system, and **cost effective**, being much cheaper and faster to construct than the actual system. According to Selic, engineering models must satisfy at least these characteristics to be useful.

Selic also points out the main functions models should have in the software engineering domain:

- **Understand** the system:
  - The structure, behavior and any other relevant characteristic of a system and its environment from a given view point.
  - Adequately separate each one of its aspects, describing them at the right conceptual level.
- Serve as a **communication mechanism**:
  - Between the different kinds of stakeholders in the system: developers, final users, maintenance personnel, etc.
  - Between the other organizations: suppliers and clients who need to understand the system in order to interoperate with it.
- **Validate** the system and its design:
  - Detect errors and anomalies in the design as soon as possible. The sooner they are detected, the easier to correct them.
  - Reason about the system, inferring properties regarding its behavior (in case of executable models than can serve as prototypes).
  - Be able to realize formal analysis over the system.
- **Guide** the implementation:
  - Serve as "maps" to build the system and allow to guide the implementation in a precise way and with no ambiguities.
  - Generate, in the most automatic way possible, both the final code and the necessary artifacts to implement, configure and deploy the system.

A key difference among a software engineer and other engineers is that in the latter the medium in which models are built is very different from the medium in which systems (buildings, bridges, aeroplanes, etc.) are built. In fact, a software model and the software being modeled share the same medium: the computer. That unique feature of software allows to define automatic transformations capable of generating implementations from higher level models, something much more expensive in other disciplines. Consequently, the purpose in MDE is the implementation of systems to be as automatable as possible, something achievable thanks to model transformations.

Metamodels

A metamodel is a model that specifies the concepts of a language, the relationships between these concepts, the structural rules that restrict the possible elements in the valid models and those combinations between elements with respect to the domain semantic rules. According to Mellor et al. [MSU+04], metamodels define the structure, semantics and constraints for a family of models. As an example, the metamodel for UML is a model that contains the elements to describe UML models, such as Package, Classifier, Class, Operation, Association, etc. The metamodel for UML also defines the relationships among these concepts, as well as integrity constraints in the models. Examples of these constraints are those that force associations to only connect classifiers, and not packets or operations.

In this way, each model is described in the language defined by its metamodel, so there is a conformance relation among a model and its metamodel. A metamodel is itself a model and, consequently, it is written in the language defined by its meta-metamodel. According to the OMG, the recursive process for defining models which conform to models at a higher level of abstraction ends when the meta-metamodel level is reached, since meta-metamodels conform to themselves, as shown in Figure 2.1.

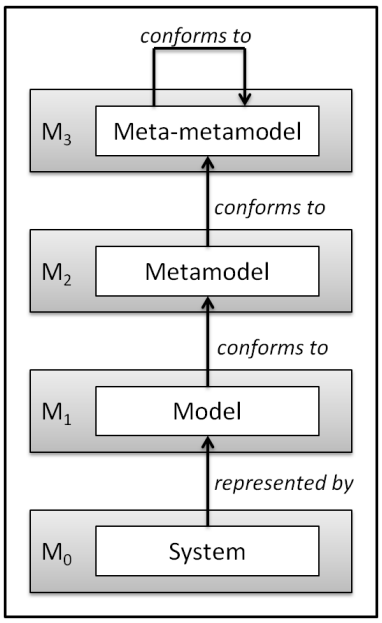


Figure 2.1: Organization in four levels proposed by the OMG.

Figure 2.2 shows a concrete instantiation of the organization proposed by the OMG. MOF (Meta-Object Facility) is the language of the OMG to describe metamodels; SPEM (Software Process Engineering Metamodel), UML (Unified Modeling Language) and CWM (Common Warehouse Metamodel) are modeling

languages conforming to MOF to describe business processes, systems and data warehouses, respectively.

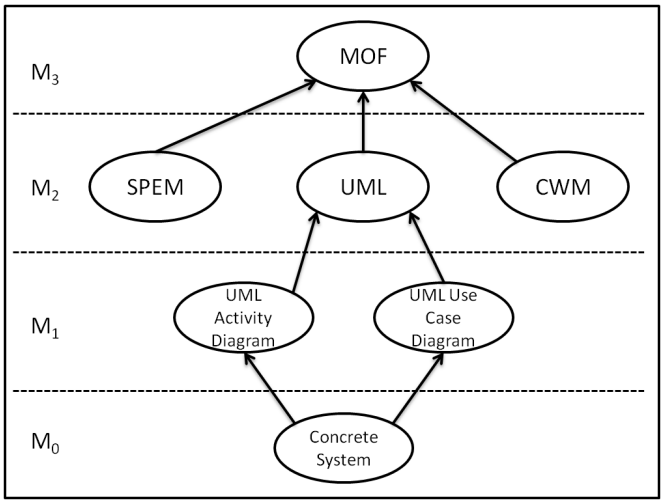


Figure 2.2: Example of the organization in four levels proposed by the OMG.

DSLs, DSMLs, DSVLs

Domain-Specific Languages (DSLs), Domain-Specific Modeling Languages (DSMLs) and Domain-Specific Visual Languages (DSVLs) are languages that deal with concepts close to the problem domain. The difference among the three terms is slight and not clearly or officially established, so we are going to give our own view of the terms.

We consider DSL as the more general term, and it involves DSML and DSVL. Both textual or visual programming languages, tailored for particular problem domains, can be considered DSLs. Examples of DSLs include HTML, Logo (multi-paradigm computer programming language used in education), Mata (for matrix programming), Mathematica and Maxima (for symbolic mathematics), SQL (for relational database queries), YACC grammars (for creating parsers), etc. Creating a DSL is worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than an existing general purpose language would allow and if the type of problem in question reappears sufficiently often.

A DSML is a kind of DSL where models (and not code) are the main artifacts. In this way, we do not consider textual DSLs as DSMLs. According to Balasubramanian et al. [BGK+06], a DSML is itself a model that defines all the possible models that developers can build using it. These languages tend to support higher level abstractions than general-purpose modeling languages, and are closer to the problem domain than to the implementation domain. Thus, a DSML follows the domain abstractions and semantics, allowing modelers to perceive themselves as

working directly with domain concepts. Furthermore, the rules of the domain can be included into the language as constraints, disallowing the specification of illegal or incorrect models. DSMLs allow domain experts express the model of their system using the vocabulary they normally use and independently from the implementation platforms.

The terms DSVL and DSML are practically synonyms. However, we are going to use the term DSVL over DSML because the languages we will define through this dissertation are mainly graphical and based in their concrete (visual) syntaxes.

When defining any DSL, its abstract syntax, concrete syntax and semantics must be clearly defined. In the following we explain each of these terms.

Abstract Syntax. The abstract syntax of a language describes the vocabulary with the concepts of the language, the relationships between them, and the rules that allow to build the sentences (programs, instructions, expressions or models) valid for the language. Such rules restrict the elements in the valid models among all those possible, as well as those element combinations that respect the domain semantic rules. **Metamodels** are the most natural way to describe the abstract syntax of a language.

Concrete Syntax. Every language (not only modeling or programming languages) has a notation that allows the representation and construction of its models. The concrete syntax of a language defines the notation used for representing the models than can be described with that language. Mainly, there are two kinds of concrete syntaxes: visual and textual.

- Visual syntaxes describe the models in a diagrammatic manner, using symbols to represent their elements and relationships among them. For example, UML offers rectangular boxes to represent classes e instances, and lines to represent associations and links. This syntax allows to represent a lot of information in an intuitive and easily comprehensible manner, which is its main advantage. On the other hand, its main drawback is that it does not allow to specify the systems with a high level of detail, since diagrams would get very complex. Besides, the expressiveness of graphical notations is normally limited. This is why they are normally complemented with specifications described with textual notations such as OCL (Object Constraint Language).

- Textual syntaxes allow to describe models using sentences composed by strings, in a similar way as programming languages. Examples are OCL and XMI (XML Metadata Interchange). The former is used to specify constraints over models or metamodels, while the latter uses XML to represent models in a textual way. HUTN (Human-Usable Textual Notation) is a textual syntax defined by the OMG to describe models and metamodels. The main advantage of textual notations is that they allow to describe complex expressions and relationships among model elements, with a higher level

of detail. However, the models described normally grow very fast and they become hard to use and uncomprehensible by humans due to their extension and complexity.

In practice, the best solution is to use a combination of graphical and textual notations, gathering the advantages of both.

Semantics. Both the abstract and concrete syntaxes describe models of a system. However, the information provided by these syntaxes may not be enough to understand the significance of the model in a precise way or to reason about it and its behavior. For this reason, the definition of a language must also provide information about its semantics, apart from its concrete and abstract syntaxes. In general, what a model *seems to represent* is not the same as what it *really means*. There are several ways to define the semantics of a system, depending on how it is to be used.

- **Denotational** semantics translate (or transform) each sentence or model of the language in a sentence or model in another language with well-defined semantics. The target language is normally a mathematical formalism (known as semantic domain). The transformation of a model specified in a language into another model in a different language is made by means of horizontal model transformations. An example of how denotational semantics are given to a DSL is shown later.
- **Operational** semantics describe the behavior of the system models in an explicit way, by means of a language based on actions or defining operations whose behavior is specified.
- **Axiomatic** semantics describe a set of rules, axioms or predicates that well-formed models must satisfy, and they interpret them in a logic where it is possible to reason about them. The Hoare logic for programming languages is an example of this kind of semantics, as well as the rewriting logic for graph grammars.

Model Transformations

As stated before, models are key parts in MDE. For that reason, it is important to count on mechanisms to manipulate them. Model transformations (MTs) are at the heart of MDE, and provide the essential mechanisms for manipulating and transforming models. MTs provide mechanisms to specify how output models are produced from input models. Czarnecki and Helen [CH06] present in Figure 2.3 an overview of the main concepts involved in a model transformation. It shows a simple scenario of a transformation with one input (*source*) and one output (*target*) model. Both models conform to their respective metamodels. The model transformation is defined with respect to the metamodels and is executed on concrete models by a transformation engine. In general, a transformation may

have multiple source and target models, and the source and target metamodels may be the same in some situations (in the so-called *in-place* model transformations).

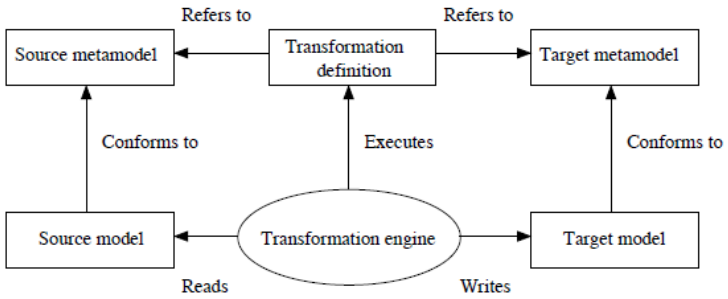


Figure 2.3: Basic concepts of a model transformation ([CH06], p. 3).

In MDE there are different types of model transformations, according to different criteria.

- Abstraction level of input and output models.
  - *Vertical* transformations. They relate models of the same system located at different levels of abstraction. They can be applied in both directions.
  - *Horizontal* transformations. They relate models at a similar abstraction level. They are normally used to keep the consistency among different models of a system. For example, they guarantee that the information modeled in a view of a system is consistent with the information of a different view of the same system and at the same abstraction level.
- Type of language used to specify the rules. There are *declarative*, *imperative* and *hybrid* (a combination of declarative and imperative) languages.
- Depending on directionality.
  - *Unidirectional* transformations. Rules are executed in only one direction, so there are clearly an input and an output models.
  - *Bidirectional* transformations. Rules can be applied in both directions, so both models are input and output.
- According to source and target models.
  - *Exogenous* transformations. Source and target metamodels are different. It is the most common case.
  - *Endogenous* transformations. Source and target metamodels are the same. These transformations are known as *in-place*.

- Type of target model.
  - *Model to model* (M2M) transformations. They generate output models from input models.
  - *Model to text* (M2T) transformations, also known as injectors. They generate text from models.

Among the many existing model transformation languages, such as QVT, ATL, Kermeta, JTL, GReAT, etc., ATL deserves an explanation since it is the one used through this dissertation.

ATL

The Atlas Transformation Language [JABK08a] (ATL) is one of the most popular and widely used model transformation languages. It is a hybrid model transformation language containing a mixture of declarative and imperative constructs. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models (Fig. 2.4). During the execution of a transformation, source models may be navigated but changes are not allowed. Target models cannot be navigated.

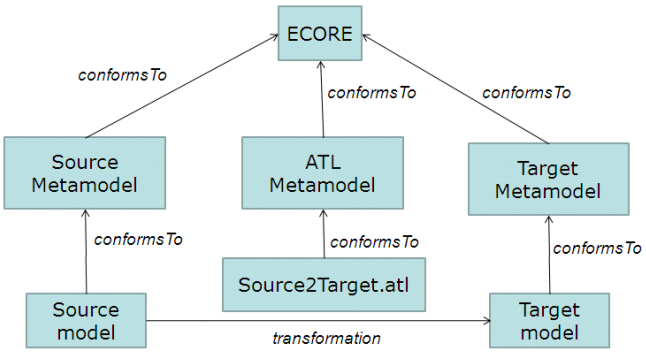


Figure 2.4: ATL model transformation schema.

ATL modules define the transformations. A module contains a mandatory header section, an import section, and a number of helpers and transformation rules. The header section provides the name of the transformation module and declares the source and target models (which are typed by their metamodels). Helpers and transformation rules are the constructs used to specify the transformation functionality.

Declarative ATL rules are called **matched rules** and **lazy rules**. Lazy rules are like matched rules, but are only applied when called by another rule. They both specify relations between source patterns and target patterns. The source pattern of a rule specifies a set of source types and an optional guard given as a Boolean expression in OCL. A source pattern is evaluated to a set of matches

in source models. The target pattern is composed of a set of elements. Each of these elements specifies a target type from the target metamodel and a set of bindings. A *binding* refers to a feature of the type (i.e., an attribute, a reference or an association end) and specifies an expression whose value is used to initialize the feature. Lazy rules can be called several times using a collect construct. **Unique lazy rules** are a special kind of lazy rules that always return the same target element for a given source element. The target element is retrieved by navigating the internal traceability links, as in normal rules. Non-unique lazy rules do not navigate the traceability links but create new target elements in each execution.

In some cases, complex transformation algorithms may be required, and it may be difficult to specify them in a declarative way. For this reason ATL provides two imperative constructs: **called rules** and **action blocks**. A called rule is a rule called by others like a procedure. An action block is a sequence of imperative statements and can be used instead of or in combination with a target pattern in matched or called rules. The imperative statements in ATL are the usual constructs for attribute assignment and control flow: conditions and loops.

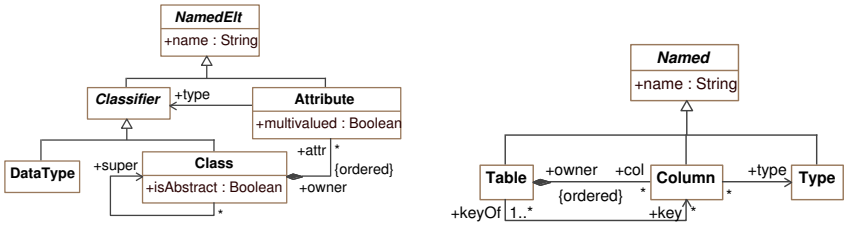
ATL also provides the **resolveTemp** operation for dealing with complex transformations. This operation allows to refer to any of the target model elements generated from a given source model element: resolveTemp(srcObj,targetPatternName). The first argument is the source model element, and the second is a string with the name of the target pattern element. This operation can be called from the target pattern and imperative sections of any matched or called rule.

ATL has two execution modes, the normal (default) execution mode and the refining one. In the former, the ATL developer has to specify, either by matched or called rules, the way to generate each of the expected target model elements. This execution mode suits to most ATL transformations where source and target metamodels are different. Using the refining mode, ATL developers can define transformations that modify the source model to obtain the target model, since both models conform to the same metamodel.

A formal semantics for ATL. The ATL language has normally been described in an intuitive and informal manner, by means of definitions of its main features in natural language. However, this lack of rigorous description can easily lead to imprecisions and misunderstandings that might hinder the proper usage and analysis of the language, and the development of correct and interoperable tools.

In our published papers [TV10, TV11] we investigated the use of rewriting logic [Mes92], and its implementation in Maude [CDE+07], for giving semantics to ATL. The use of Maude as a target semantic domain brings very interesting benefits, because it enables the simulation of the ATL specifications and the formal analysis of the ATL programs. In particular, we showed how our specifications can make use of the Maude toolkit to reason about some properties of the ATL rules.

Although this is not the goal of this dissertation, let us show a simple example



(a) Source metamodel. (b) Target metamodel.

Figure 2.5: Transformation metamodels.

Listing 2.1: DataType2Type ATL matched rule

```
rule DataType2Type {
  from
      dt : Class!DataType
  to
      t : Relational!Type (
          name <- dt.name
      )
}
```

of how semantics based on rewriting logic are given to a small ATL rule. Among the semantics described before, these are *denotational*. The metamodels for the transformation are shown in Figure 2.5. The purpose of the transformation rule is very simple. For each DataType instance, a Type instance has to be created. The names of both instances have to correspond. The ATL rule is shown in Listing 2.1. The translation of this ATL rule into Maude, i.e., its denotational semantics given in the formal language Maude, is shown in Listing 2.2.

Listing 2.2: Encoding in Maude of the DataType2Type ATL matched rule

```
crl[DataType2Type] :
  Sequence[
  (@ClassMm@ {
  < DT@ : DataType@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ }) ;
  (@RelationalMm@ { OBJSETTT@ })
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < DT@ : DataType@ClassMm | SFS >
    OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm :
        VALUE@CNT@ + 2 >
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ DT@ ] #
        trgEl@TraceMm : Sequence[ T@ ] #
        rlName@TraceMm : "DataType2Type" #
```

```
        srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
  OBJSETT@}) ;
  (@RelationalMm@ {
  <  T@ : Type@RelationalMm | name@Named@RelationalMm : << DT@ .
        name@NamedElt@ClassMm ; CLASSMODEL@ >> >
  OBJSETTT@ })
  ]
  if
  CLASSMODEL@ := @ClassMm@ {
    < DT@ : DataType@ClassMm | SFS >
  OBJSET@ } /\
  TR@ := newId(VALUE@CNT@) /\ T@ := newId(VALUE@CNT@ + 1) /\
  not alreadyExecuted(Sequence[DT@], "DataType2Type", @TraceMm@
    { OBJSETT@ })
.
```

Of course, in order to translate ATL rules into Maude code, we also have to represent models and metamodels in Maude. The interested reader can consult [TV10, TV11].

2.2 The *e-Motions* Language

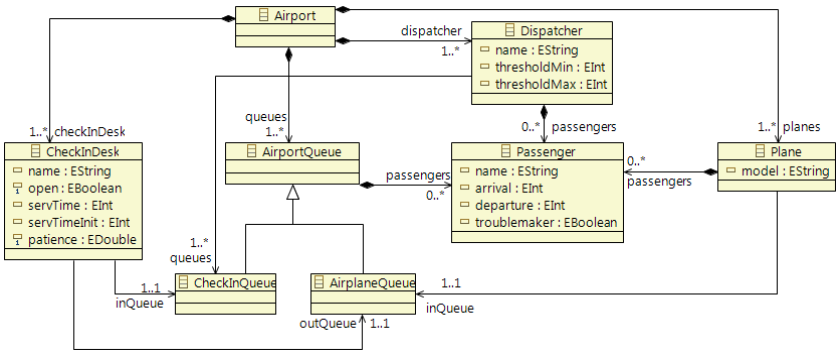
This section describes the *e-Motions*' features that will be used throughout his document. We present as well an implementation example to clarify all of them.

e-Motions is a language and tool that extends in-place model transformations with a model of timed behaviors and a mechanism to state action properties. It has been created as a plugin for the Eclipse 3.4 (Ganymede) version. Real-time domain-specific visual languages (DSVLs) can be specified in a graphical and intuitive way, and simulation, reachability and model-checking analysis can be performed afterwards. In order to define a DSVL in *e-Motions*, first the metamodel of the system must be defined. It describes the structure of the system and conforms the abstract syntax. Then, a visual concrete syntax is defined, since *e-Motions* allows to associate every metamodel class with a graphical element. It conforms the graphical syntax.

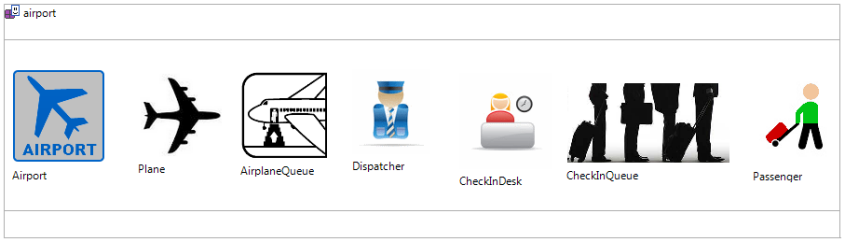
In the example we are going to model the check-in and boarding process in an airport. This is, we are going to model the arrival of passengers, how each one is assigned to check-in desk's queues, how they do the check-in, and how passengers eventually get on the plane. The metamodel for this example is the one shown in Figure 2.6(a). The Airport is composed of Dispatchers, AirportQueues of different type (CheckInQueue and AirplaneQueue), desks for doing the check-in (CheckInDesk), Planes, and the Passengers who are in the airport. Each Dispatcher has associated one or more CheckInQueue with the queue relationship. Dispatchers have Passengers associated too. The aim of Dispatchers is to welcome Passengers to the airport and indicate them which CheckInQueue they have to go to in order to wait for the check-in. The queues in the airport, objects of type AirportQueue, contain a sequence of Passengers (passengers containment relationship). Objects of type CheckInDesk aim to realize the chek-in of Passengers. This is, the first Passenger in the CheckInQueue of a CheckInDesk will

do the check-in in such CheckInDesk. Once he/she is done, the next Passenger will realize the check-in. When Passengers are done with the check-in, they go to an AirplaneQueue to later get on a Plane.

Regarding the attributes in every class, the Dispatcher has a name and two thresholds for deciding when to open or close desks. Each CheckInDesk has a name and an open boolean attribute that indicates if that desk is currently open. The patience attribute is an indicator of how fast the person behind the desk loses patience: the higher the value, the faster he/she loses patience and the slower he/she will serve passengers as time goes by. Attribute servTimeInit indicates the initial time in serving a passenger, and servTime is the time spent in serving a passenger as time goes by, and it is related to the patience rate. Passengers have a name and an attribute that indicates if they are trouble makers or not. If they are, then they take more time in being served in the CheckInDesks than a regular Passenger. Finally, Planes contain information of their model.



(a) Abstract syntax.



(b) Concrete syntax.

Figure 2.6: Abstract and concrete syntaxes for the check-in and boarding process example.

The visual concrete syntax for our system, i.e., the image associated to each non-abstract class in the metamodel, is the one shown in Figure 2.6(b). The last step is to define the semantics in terms of behavioral rules, as we explain in Section 2.2.2.

2.2.1 The *e-Motions* Behavioral Metamodel

In *e-Motions*, the behavior of a DSVL is specified by a set of in-place rules and an optional set of helpers. Such behavior conforms its semantics, *operational* semantics according to the classification given in Section 2.1.2. The *e-Motions* behavioral metamodel is shown in Figure 2.7. There are two types of rules to specify time-dependent behavior, namely *atomic* and *ongoing* rules. The former represent actions with a specific duration, while the latter represent actions that progress continuously with time. Helpers are expressed as OCL operations and are library functions useful for specifying some of the parameters or object attributes. They can be used throughout the behavioral specifications.

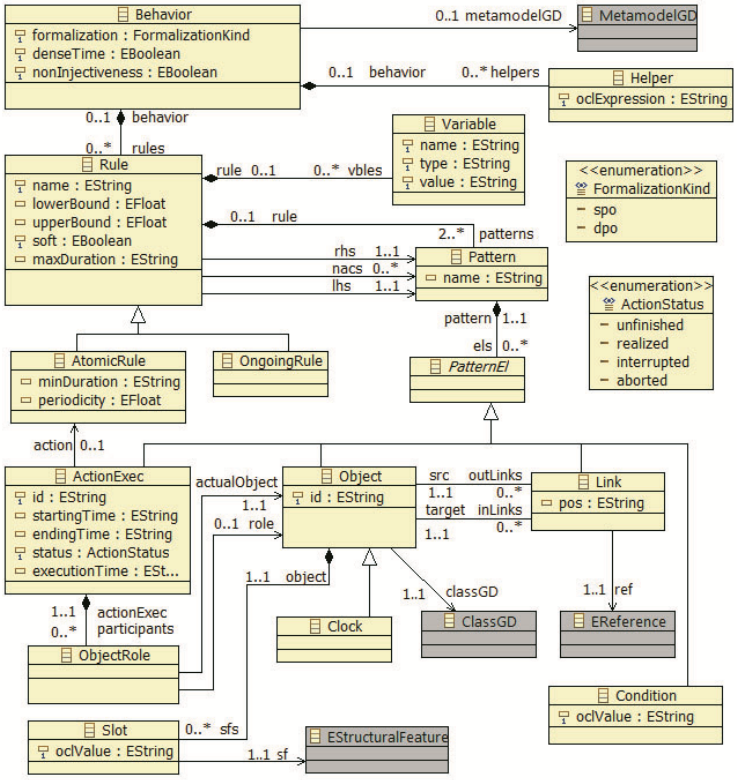


Figure 2.7: The *e-Motions* metamodel.

Rules are composed of patterns: one left-hand side (LHS), one right-hand side (RHS) and zero or more negative application conditions (NACs). They also may contain a set of variable definitions. The context of such variables is the rule where they are defined, and their values are computed when the rule is triggered, only once. Rule patterns may contain objects, links and action execution elements (ActionExec metaclass) that represent action occurrences. LHS and NAC patterns may also include conditions. Objects can be composed of slots

(structural feature-value pairs). When an object is placed in a LHS or NAC pattern, its slots represent attribute conditions. When it is placed in a RHS pattern, its slots represent attribute computations. Both slots values and conditions are specified with OCL expressions. There is also a special kind of object, named Clock, that represents the current global time elapse.

The Model of Time Used

As mentioned before, there are two kinds of rules, *atomic* and *ongoing* rules, and a special kind of object named *Clock* that represents the current global time elapse. In the following we explain in more detail each of them.

Atomic Rules

Atomic rules are defined as in-place transformation rules of the form $l : [\text{NAC}]^* \times \text{LHS} \overset{[t,t']}{\rightarrow} \text{RHS}$, where $[t,t']$ expresses the duration interval of the action modeled by the rule, i.e., the minimum and the maximum amount of time needed to perform the action. As in the case of normal in-place transformation rules, an atomic rule can be *triggered* whenever an occurrence (match) of its LHS, and none of its NAC patterns, is found in the model. Then, the action specified by such rule is scheduled to be *realized* between t and t' time units later. At that time, the rule is applied by substituting the match by its RHS and performing the attribute computations.

Since actions have durations, the same model elements can be involved in two different rules at the same time. For this reason, *e-Motions* offers mechanisms to abort some rules whose execution had started at some point. In this way, the only condition for the final application of an atomic rule is that the elements involved are still present, otherwise the action is *aborted*.

Standard in-place rules where there is no specification of the time they consume are modeled by *instantaneous* rules. They are atomic rules whose execution takes zero time units. No other action can occur during the execution of an action of this type.

Ongoing Rules

They model actions that are continuously progressing and require to be continuously updated. An example is an action that models the consumption of a phone battery, whose level decreases continuously with time. These rules have a maxDuration attribute which specifies the time limit of the rule. In this way, ongoing rules are used to model actions that a) do not have *a priori* duration time—they progress with time while the rule preconditions (LHS and NACs) hold, or until the time limit (maxDuration) of the action is reached—and b) are required to be continuously updated—their effects are always computed before the triggering of any atomic rule.

At this point the reader may be wondering if ongoing rules, since they model actions that need to be continuously updated, are executed every time unit. The answer is negative: the effects of ongoing actions are always realized before the triggering of atomic rules. This solution optimize the performance of the simulations, since the elements present in ongoing rules are updated only when strictly necessary.

Global Time Elapse

A special kind of object, named *Clock*, is provided. It represents the current global time elapse. A unique and read-only *Clock* instance is provided by the system to model time elapse through the underlying platform. This object can be used by designers in their timed rules to get the current time. It will be very useful when modeling the non-functional properties of systems in our proposal.

Action Statements

In order to be able to model both state-based and action-based properties, *e-Motions* implements a reflective mechanism that allows extending model patterns with *action executions* to specify action occurrences. These action executions specify the type of the action (i.e., the name of the atomic rule), its status (i.e., if the action is unfinished, realized, interrupted or aborted) and its identifier. They may also specify its starting and ending time, its execution time, and the set of participants involved in it.

Action executions whose status is unfinished refer to actions that are currently happening, i.e., actions which have been triggered but whose duration time has not been consumed yet. The status realized refers to actions that were already performed, i.e., the effects specified in their RHS were already applied. Action executions whose status is interrupted refer to actions that were triggered but not realized because their corresponding action executions were deleted by a rule explicitly modeled by the user for that purpose. Finally, status aborted refers to actions that were triggered but not realized because some of its participants were removed from the system before the duration time of the action was consumed, i.e., actions that were automatically discarded because they could not be completed.

2.2.2 Illustrative example

After having defined the abstract and concrete syntaxes of our example at the beginning of Section 2.2, we have to specify its semantics, by means of a set of behavioral rules.

The first step when modeling the behavior of a system is to set the initial model. Such model, conforming to the system's abstract syntax, will be evolving as the behavioral in-place rules are triggered over it. In *e-Motions*, the initial model can be defined with an instantaneous atomic rule or with the Eclipse tree-view. In this

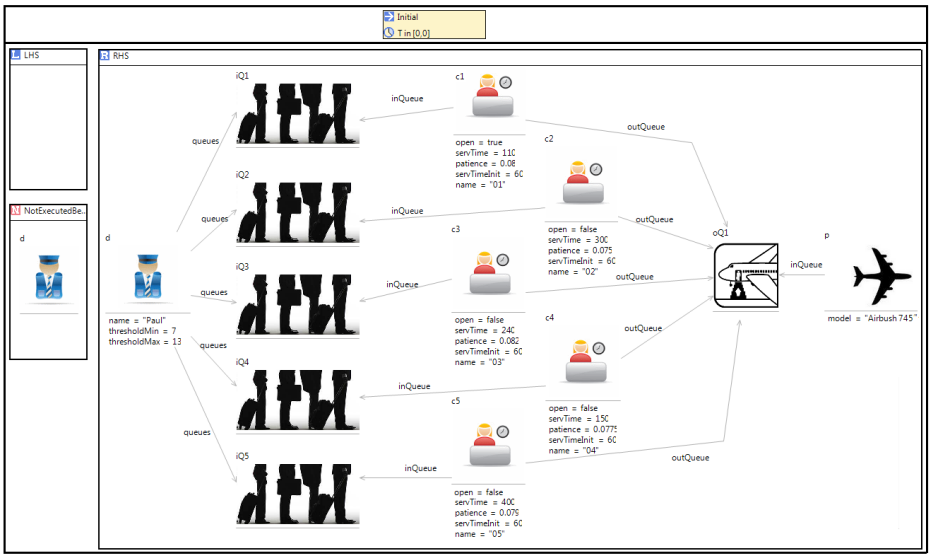


Figure 2.8: Initial rule.

case we have chosen the first option. Such rule is shown in Fig. 2.8. The LHS of the rule is empty because there is no existing model before this rule is triggered. Several objects are created in the RHS: a Dispatcher, that will be welcoming passengers to the airport; five CheckInQueues, where passengers will be sent by the Dispatcher; five CheckInDesks, where passengers in the previous queues realize the check-in; an AirplaneQueue, where passengers wait after doing the check-in for getting on a plane; and the Plane where the passengers will finally get on. There is also a NAC in this rule, which forbids its execution if there is already an instance of the class Dispatcher in the model. It means that once a Dispatcher is created, this rule is no longer triggered (as long as at least one object of type Dispatcher remains in the system). Notice that only one desk is open at the beginning. They all start with the same service time, 60—we consider for this example that time units are seconds. Each desk has a patience rate. The name and thresholds for the Dispatcher are initialized as well.

Atomic rules in Figures 2.9, 2.10, 2.11 and 2.12 model the process that passengers follow since their arrival in the airport until they get on their plane. Rule NewPassenger models the arrival of a Passenger to the Airport every 90 seconds. This rule only needs the presence of a Dispatcher to be launched (rule's LHS). In the rule's RHS we can see how the Passenger is now associated to the Dispatcher. Since the passengers relationship was defined as ordered, the Passenger is placed at its end. A variable, troubleProb, is used to determine whether the Passenger is trouble maker or not. This variable takes a value between 1 and 100 (this is achieved by the *e-Motions*' internal random function). The Passenger will be trouble maker if the remainder of dividing troubleProb by

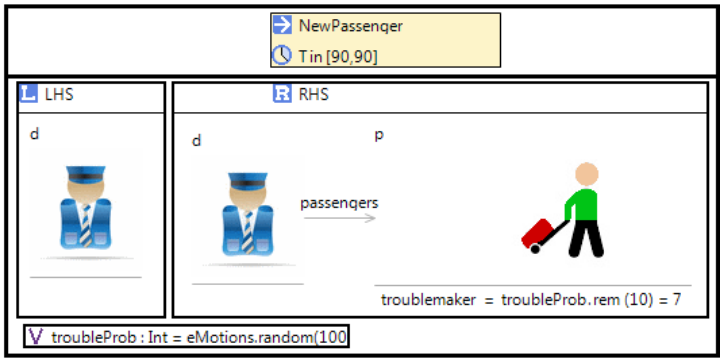


Figure 2.9: NewPassenger rule.

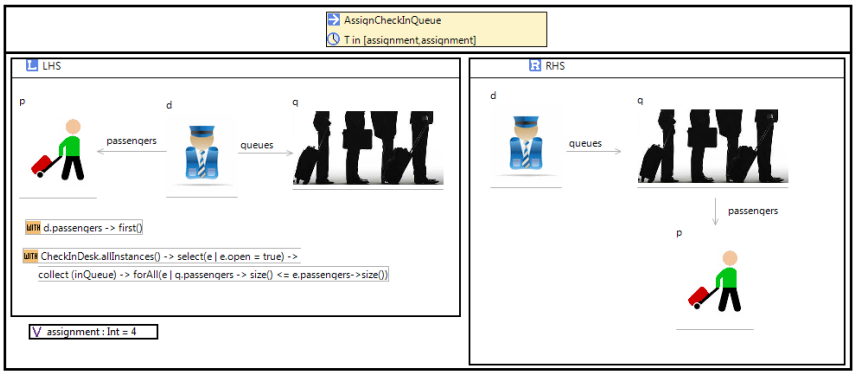


Figure 2.10: AssignCheckInQueue rule.

10 is 7. Since the probability of getting a 7 as the reminder when dividing a random number by 10 is one out of ten, we are modeling that 10% of the Passengers will be trouble makers.

Rule AssignCheckInQueue models the assignment of a Passenger to a CheckInQueue by the Dispatcher. As stated in the OCL constraints in the rule's LHS, the Passenger must be the first in the passengers relationship (first constraint). Furthermore, according to the second constraint, the CheckInQueue in the LHS is the one with the smallest number of passengers among those queues whose desk is open. In the rule's RHS the passenger has been assigned to that queue. In this rule we show how duration in rules can be specified with a variable. In this case, the assignment variable establishes the duration of the AssignCheckInQueue rule.

Rule CheckInPassenger models the check-in of the first Passenger in the CheckInQueue for a given desk. The OCL constraint in the rule's LHS indicates that the Passenger must be the first in such queue. In the rule's RHS the Passenger is placed in an AirplaneQueue, modeling that he/she realized the check-in and moved to his/her plane's queue. The duration of this rule is given by the

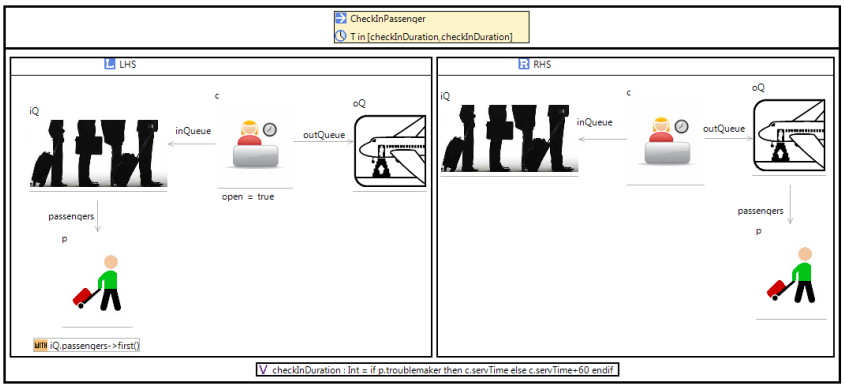


Figure 2.11: CheckInPassenger rule.

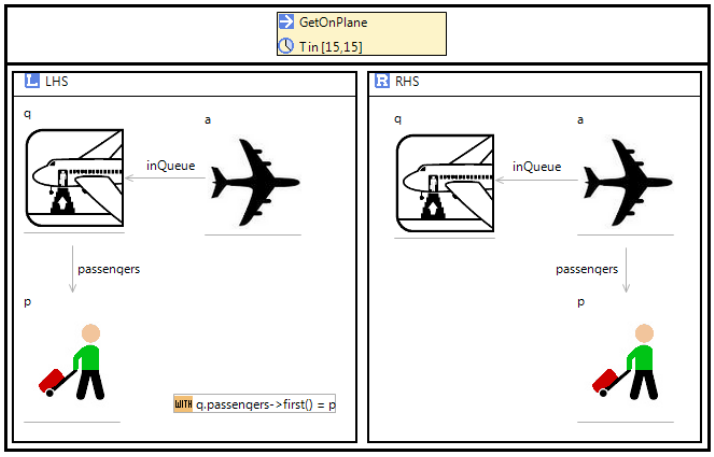
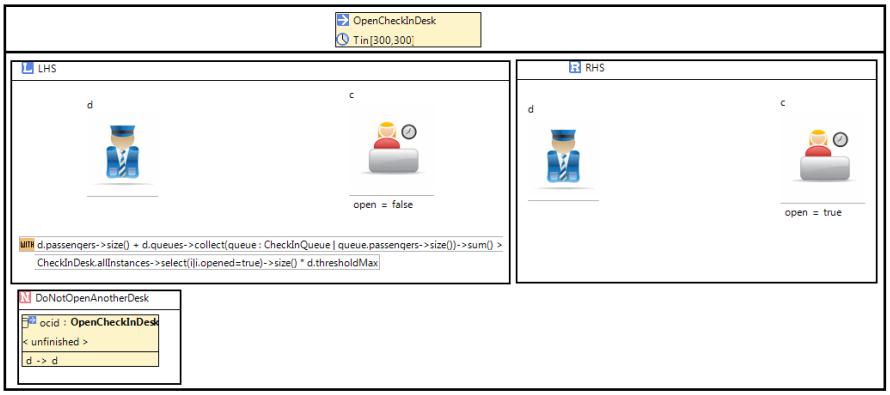


Figure 2.12: GetOnPlane rule.

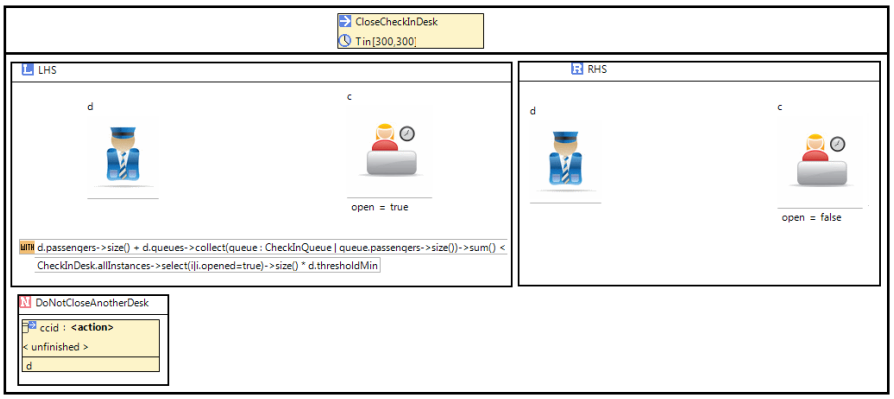
checkInDuration variable. The value of this variable is the service time of the CheckInDesk if the Passenger is not a trouble maker, and this duration plus one minute if he/she is. The way the servTime attributes of desks vary is explained later, in rule LosePatience.

Rule GetOnPlane models how the first Passenger in the AirplaneQueue gets on the Plane. The OCL constraint in the LHS forces the rule to do the matching with the first passenger in the queue. In the rule's RHS, the Passenger is not associated to the queue anymore, but to the Plane, modeling that he/she is inside it.

Rules in Figure 2.12 model the opening and closure of a CheckInDesk by the Dispatcher. The OpenCheckInDesk rule's LHS shows that the desk doing the matching must be closed. The OCL constraint forces the rule to be triggered if the number of passengers who have not done the check-in yet (i.e., the number of passengers associated with the dispatcher and those waiting in desks' queues)



(a) OpenCheckInDesk rule.



(b) CloseCheckInDesk rule.

Figure 2.13: Rules for open and close CheckInDesks.

is larger than the product between the number of open desks and the value of the thresholdMax attribute—this is the airport policy for opening a new check-in desk. In the rule's RHS, the desk has been opened. It takes 5 minutes (300 seconds) in opening a desk. There is also a NAC (DoNotOpenAnotherDesk) whose purpose is to avoid the parallel opening of a different desk. For this, an action execution is placed within the NAC. The type of the action is the same as the rule it is in, OpenCheckInDesk, and its status is unfinished. There is also a participant: the Dispatcher of the rule. Summing up the purpose of this NAC, it forbids the triggering of this rule if there is another action of this type in execution (i.e., the rule OpenCheckInDesk is already being executed) with the same Dispatcher as the one this rule is doing the matching with. Notice that if there is only one Dispatcher, this rule can only be fired once at a time. However, if there are more than one, there can be more than one executions of this rule at the same time. The behavior of the OpenCheckInDesk is very similar.

Finally, we model with an ongoing rule how people behind check-in desks

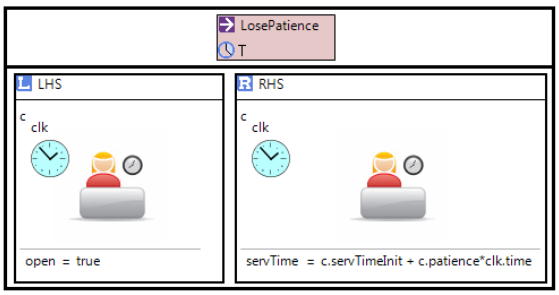


Figure 2.14: LosePatience rule.

lose patience. It is the LosePatience rule (Fig. 2.14). This rule keeps the service time of desks updated. The value of such attribute is the desk's initial service time plus the product of the desk's patience rate and the time elapsed in the system. This rule is only applied over open desks.

3

Model-Driven Performance Analysis of Rule-Based DSVLs

Domain-specific visual languages (DSVLs) play a crucial role in Model-Driven Engineering (MDE) for representing models and metamodels. The benefits of using DSVLs is that they provide intuitive notations, closer to the language of the domain expert and at the right level of abstraction. In other words, they provide languages that help model systems in a way which is closer to the problem domain. The MDE community's efforts have been progressively evolving from the specification of the structural aspects of systems to the development of languages that allow for modeling their behavioral dynamics. Thus, several proposals already exist for modeling the structure and behavior of systems. Some of these proposals also come with supporting environments for animating or executing the specifications, based on the transformations of the models into other models that can be executed [dLV06, dLV08, EHC05, EE08, EHKZ05b].

The correct and complete specification of a system includes, however, other aspects beyond structure and basic behavior. In particular, the specification and analysis of its non-functional properties, such as QoS usage and management constraints (performance, reliability, etc.), is critical in many important distributed application domains including embedded systems, multimedia applications and e-commerce services and applications.

In order to fill this gap, in the last few years researchers have faced the challenge of defining quantitative models for non-functional specification and validation from software artifacts [BMIS04, Zsc10]. Several methodologies have been introduced, all sharing the idea of annotating software models with data related to non-functional aspects, and then translating the annotated model into a model ready to be validated [CMI07, FBV+09, FJA+09]. Most of these proposals for annotating models with QoS information exist for UML-based notations, with UML Profiles such as UML-QoSFT, UML-SPT or MARTE [OMG04, OMG05, OMG08]. Several tools already exist for analyzing and simulating such models and for carrying out performance analyses.

Although these profiles provide solutions for UML models, the situation is different when domain-specific visual languages are used to specify a system. In the first place, the QoS annotations are normally written using languages which are

completely alien to system designers, because such languages are typically influenced by the analysis methods that need to be used, and written in the languages of these methods. Besides, their level of abstraction is normally lower than the DSVL notations used to specify the system, and tend to be closer to the solution domain than to the problem domain (in contrast with the problem-domain orientation of DSVLs). Furthermore, when several properties need to be analyzed, the annotated models become cluttered with a plethora of different annotations and marks (see, e.g., many of the diagrams shown in the MARTE specification [OMG08]). Another problem is that current proposals for the specification of these kinds of properties tend to require skilled knowledge of specialized languages and notations, which clashes with the intuitive nature of end-user DSVLs and hinders its smooth combination with them. Finally, most of these proposals specify QoS characteristics and constraints using a *prescriptive* approach, i.e., they annotate the models with a set of requirements on the behavior of the system (response time, throughput, etc.) and with constraints on some model elements, but are not very expressive for describing how such values are dynamically computed or evolve over time.

In this chapter we present an alternative approach to specify the properties that need to be analyzed, integrating new objects in the specifications that allow them to be captured. Our proposal is based on the *observation* of the execution of the system actions and of the state of its constituent objects. We use this approach to simulation and analysis in the case of DSVLs that specify behavior in terms of rules (which describe the evolution of the modeled artifacts along some time model), and illustrate the proposal with the running example of a production line system. After presenting the whole approach, we describe our packet switching case study. We show how, given an initial specification of the system, the use of observer objects enables the analysis of some of the properties usually targeted by performance engineering, including throughput, mean and max cycle-time for produced items, busy and idle cycles for the machines in the system, mean-time between failures, etc. One of the benefits of observers is that they can be specified in the same language that the domain expert is using for describing the system. Finally, we show how this approach also enables the specification of other important features of systems, such as the automatic reconfiguration of the system when the value of some of the observed properties change.

The structure of this chapter is the following. Section 3.1 briefly presents the running example that will be used throughout the chapter to illustrate our approach. Section 3.2 introduces the main concepts of our proposal and how they can be used to specify the system parameters that we want to analyze. Section 3.4 explains the extension of the *e-Motions* time modeling with the inclusion of probability distributions. Then, Section 3.3 shows how the specifications produced in Section 3.2 can be used to analyze the performance of the system and the tool support available for that. Once the main ideas of our proposal have been illustrated using a running example, Section 3.5 describes the general methodology

that we propose for the performance analysis of systems specified with rule-based domain-specific languages such as *e-Motions*. It also explains the current tool support and how the simulations are conducted to obtain the analysis results. Section 3.6 models another example named *packet switching*. It shows that an extensive performance analysis can be performed with our approach. Finally, Section 3.7 summarizes this chapter.

3.1 Production Line System Case Study

One way of specifying the dynamic behavior of the models expressed with a DSVL is by describing the evolution of the modeled artifacts along some time model. In MDE, this can be done using model transformations supporting in-place updates [CH03]. The behavior of the system is then specified in terms of the permitted actions, which are in turn modeled by the model transformation rules. The kind of rules used in our approach to specify this behavior are those presented in Section 2.2.

Here we present our running example of a hammer production line system (PLS). As any other DSVL and as explained in Section 2.1.2, our production line system is defined in terms of three main elements: abstract syntax, concrete syntax and semantics. The abstract syntax defines the domain concepts that the language is able to represent, and is defined by a metamodel. The concrete syntax defines the notation of the language, and in our example it is defined by assigning an icon to each concept in the metamodel. The semantics describe the meanings of the models represented in the language, and in case of models of dynamic systems (such as ours) the semantics of a model describe the effects of executing that model. In our case, semantics are specified by a set of behavioral rules.

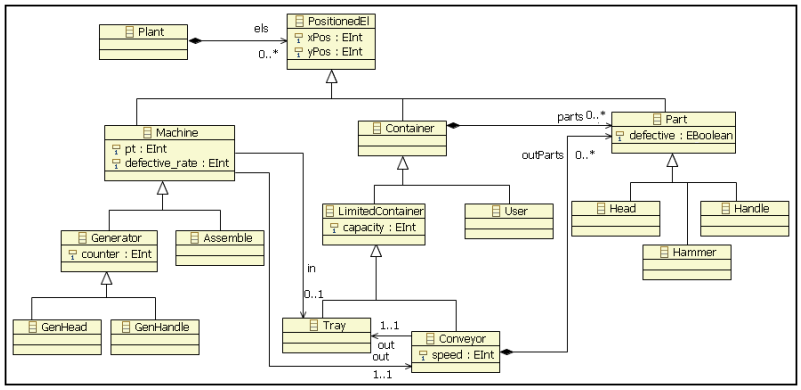


Figure 3.1: Production Line Metamodel.

The metamodel for the system is depicted in Fig. 3.1. There are different kinds of Machines (head generators—GenHead—, handle generators—GenHandle—, and Assemblers), Containers (Users and containers with a limited capacity, such

as Trays and Conveyors) and Parts (Heads, Handles and Hammers). They all have a position in the plant, indicated by a set of coordinates. Generators will produce as many Parts as their counter indicates and deposit them on Conveyors; these move Parts from Machines to Trays at a given speed; and Assemblers consume Parts from Trays to create Hammers, which are deposited on Conveyors and finally collected by operators. Parts can be either defective or not. Machines work according to a production time (pt) that dictates the average number of time units that they take to produce a Part. They also have an attribute (defective_rate) that determines the percentage of defective Parts they produce, which depends on their production time (normally, the faster they work the more defective Parts they produce). Trays and Conveyors can contain Parts up to their capacity.

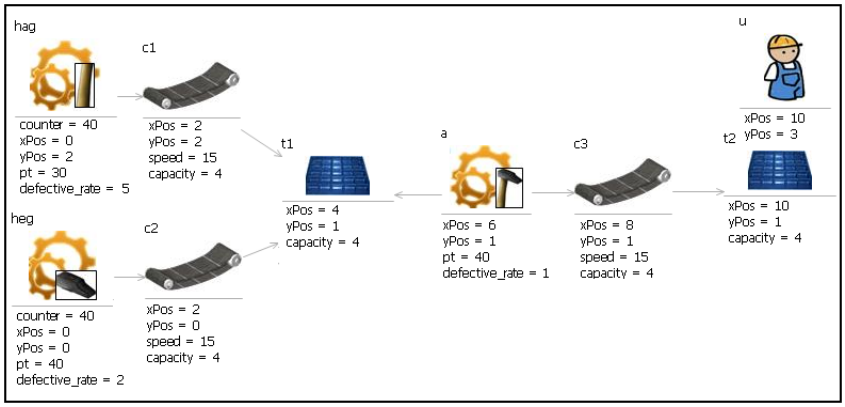


Figure 3.2: A model of the system, which will be used as initial configuration.

Fig. 3.2 shows an example of a model of the system, depicted with *e-Motions*, with a set of objects and values for their attributes, which will be used as initial configuration. In that Figure, we can see the concrete syntax given to the concepts of the production line metamodel.

As explained in Section 2.2, the behavior of the system is then expressed in terms of a set of rules, each one representing a possible action. In the case of our PLS, its behavior can be described by 6 rules: one for generating Handles (GenHandle); one for generating Heads (GenHead); one for describing how Conveyors move Parts (Carry); one for assembling Heads and Handles into Hammers (Assemble); one for depositing Parts in Trays once they have been transported through Conveyors (Transfer), and one final rule to describe how an User collects assembled Hammers (Collect).

For example, Fig. 3.3 shows the atomic Carry rule, which specifies how a Part is transported through a Conveyor, moving it from the beginning of the Conveyor to its end. In the rule's LHS, the Part is related to the Conveyor with the parts reference, indicating that the Part is placed at the beginning of it. In the rule's RHS, the relation between both objects is outParts, which indicates that the Part has been placed at the end of the Conveyor and is ready to be transferred to the

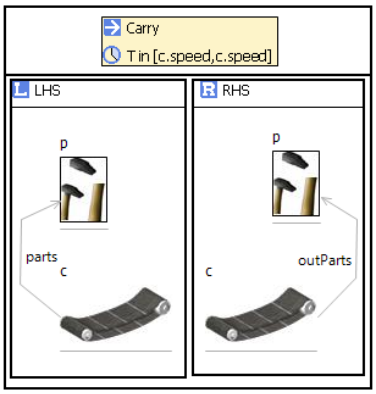


Figure 3.3: Carry Rule.

Tray connected to it. The time this rule spends, which simulates the time needed to move a Part through a Conveyor, is the corresponding speed of the Conveyor (15 time units in our example).

3.2 Monitoring the System with Observers

Once we can count on languages for specifying models and their behavior, the next step is to analyze their non-functional properties. For that we need to be able to express the properties that we want to analyze (e.g., delays, mean-time between failures, or end-to-end throughput), and then we need to have a simulation engine that executes the specifications.

3.2.1 Specifying the Properties to be Analyzed

Although the number of non-functional properties that can be defined for a system can be large, in this chapter we will concentrate on some representative QoS properties [ISO99]. Let us suppose then that we want to analyze the following performance parameters of the PLS:

- **Throughput**. Number of non-defective Hammers collected by the operator per unit of time. It is calculated with the formula $th = h/t$, where h is the number of non-defective collected Hammers and t is the time the system has been working.
- **Mean time between failures (MTBF)**. It is the mean time between producing defective Hammers. This value is given by the formula $MTBF = t/d$, where t is the time the system has been working and d is the number of defective Hammers.
- **Idle-time**. Amount of time that a Machine has not been working (idle, waiting for Parts or waiting for the output Conveyor to be free).

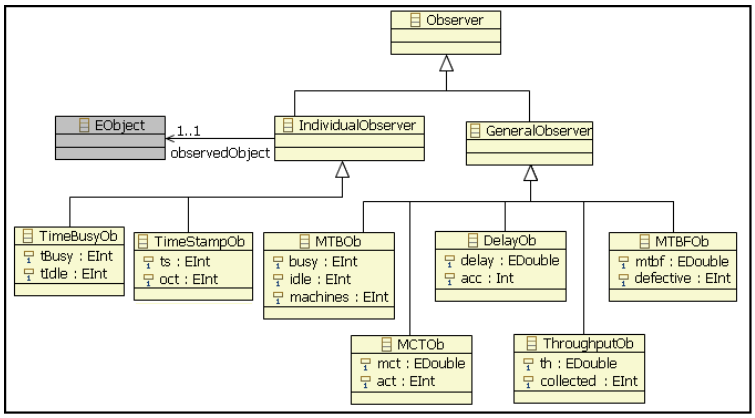


Figure 3.4: Observers Metamodel.

- **Mean Idle-time**. Average idle time of all Machines.
- **Cycle time**. Time required to produce a Hammer: from the production of its Parts until its final collection.
- **Mean cycle time (MCT)**. Mean cycle time of all produced Hammers. This is computed as $MCT = ct/H$, where ct is the addition of cycle times of every collected Hammer and H is the number of all collected Hammers.
- **Delay**. This property indicates the difference between the *theoretical* optimal cycle time of a Hammer and its actual cycle time. This is, supposing that every collected Hammer has an optimal production time, this property will indicate the average delay of their cycle time with respect to this optimal production time. The delay of a single Hammer is calculated by subtracting its optimal cycle time from its real cycle time.

Let us clarify here that all collected Hammers will be taken into account for computing the system mean cycle time and delay, while only non-defective ones will be considered for computing the system throughput.

3.2.2 Defining the Observers' Structure

To specify and calculate the value of system properties, we propose the use of *observers*. An observer is an object whose purpose is to monitor the execution of the system: the state of the objects, of the actions, or both. We will use two kinds of observers depending on whether they monitor the state of the complete system or the state of individual objects. In the former case, observers are created with the system and remain there throughout its entire life. In the latter case, observers are created and destroyed with the objects they monitor.

Observers, as any other objects, have a state and a well-defined behavior. The attributes of the observers capture their state and are used to store the variables that

we want to monitor. We have defined an *Observers metamodel*, which is shown in Fig. 3.30. It defines two kinds of observers, IndividualObservers and GeneralObservers, both inheriting from a general Observer class. Observers of individual objects have a reference to an EObject class, which is the interface implemented by every model object in the *Eclipse Modeling Framework* (EMF [Ecl]). In this way, these observers can be associated to any element of any model. We count on two specific observers for individual objects and five for the whole system. Let us explain the objectives of each of them for our example:

- **TimeBusyOb**. It monitors the time a Machine is working and the time it is idle.
- **TimeStampOb**. It monitors individual Parts. Its ts attribute stores the time at which the generator started producing the Part. Attribute oct keeps the *optimal cycle time* of the Part. It is updated as the Part moves forward in the system. It is used to compute the delay.
- **MTBOb**. This observer monitors the average working time of all the Machines in the plant. It uses the number of Machines (attribute machines) as well as the attributes of the TimeBusyOb observers of all Machines.
- **MCTOb**. This observer keeps the mean cycle time of every collected Hammer in its mct attribute.
- **DelayOb**. It monitors the average delay of all produced Hammers. The sum of the delays of all collected Hammers is stored in its acc attribute, while the Hammers' average delay is stored in its delay attribute.
- **ThroughputOb**. It stores the throughput of the system in its th attribute. The number of collected Hammers is stored in its collected attribute.
- **MTBFOb**. It records the mean time between failures in the system and the number of defective Hammers.

Note how the use of observer objects provides a modular approach to extend the state of the system with the attributes that capture the properties we want to analyze. However, to conduct the performance analyses we also need to specify the behavior of the observers while the system executes. Note as well that despite dealing with performance properties here, we have also included mean time between failures, what is considered a reliability property. In the next chapter we study this property with more detail.

3.2.3 Defining the Observers' Behavior

The idea of analyzing the system with observers is to combine the original metamodel (Fig. 3.1) with the Observers' metamodel (Fig. 3.30) to be able to use the observers in our DSVL for specifying production line systems. *e-Motions* allows

users to merge several metamodels in the definition of a DSVL behavior. This is, we can define the *Observers metamodel* in a non-intrusive way, i.e., we do not need to modify the system metamodel to add observers to it. Furthermore, this approach also enables the reusing of observers across different DSVLs.

The behavior of the observers is also specified using rules. In fact, their behavior is described by enriching the system rules with information about the observers—so that now the rules not only describe how the system behaves but also how the observers monitor the system and update their state. In what follows we show how the behavior of the observers is specified by adding them to the system rules.

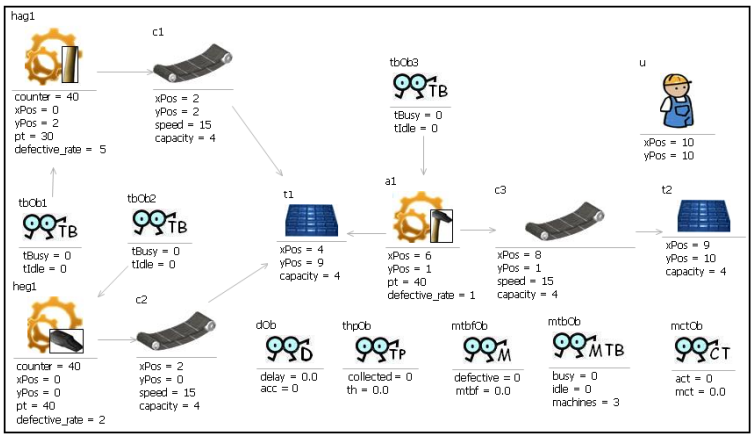


Figure 3.5: The initial configuration of the system with Observers.

Firstly, we modify the model with the initial configuration of objects by adding observers to it (Fig. 3.5). We have defined five observers for the whole system (DelayOb, ThroughPutOb, MTBFOb, MTBOb and MTCOb), and three individual TimeBusyOb observers, each one associated to a Machine to record the time it is working.

Then, we need to specify the behavioral rules. The first two define how and when Heads and Handles are generated. Fig. 3.6 shows the GenHandle atomic rule that generates a new Handle every time it is launched. The time it spends in the generation of a Handle depends on its production time attribute (pt). Instead of a fixed time, we use a random value from the interval $[hg.pt-3, hg.pt+3]$ to specify its duration. It uses the $random(n)$ function available in *e-Motions* that returns an integer value between 0 and n. For this rule to be applied, the LHS pattern indicates that the system has to have a HandleGen generator which has to be connected to a Conveyor. The LHS pattern also has a condition (see the WITH clause), so the rule can only be applied if the attribute counter of the GenHandle object is greater than 0 and the Conveyor has room for the generated Part. In the RHS pattern a new Handle is generated and it acquires the position of the Conveyor connected to the HandleGen machine. To decide whether the

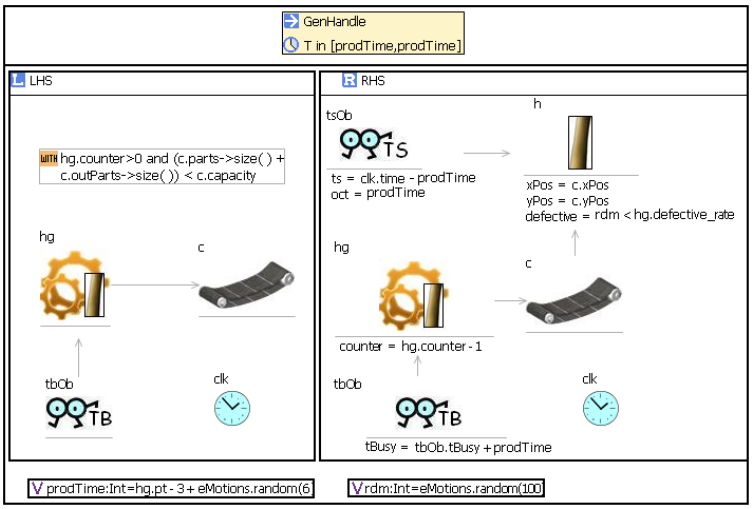


Figure 3.6: GenHandle Rule.

new Handle generated is a defective Part or not, we check if a random variable between 0 and 100 (rdm) is smaller than the defective rate of the generator. The counter value of the GenHandle is decreased in 1 unit, which represents the fact that a new Handle has been produced. A TimeStampOb observer is created and it is associated with the Handle, storing in its ts attribute the time at which the HandleGen started to generate the Handle—we use the Clock instance to get the time the system has been working. Note the use of OCL to compute the values of the objects' attributes. Analogously, the GenHead rule (not shown here) models the generation of Heads.

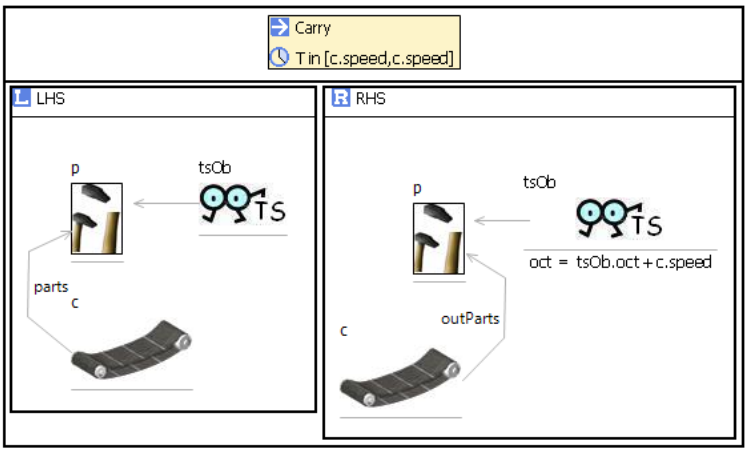


Figure 3.7: Carry Rule with Observer.

As soon as a new Part is placed on a Conveyor, it is transported from its be-

ginning to its end. We modify the Carry rule shown in Fig. 3.3, which models this behavior, to introduce the corresponding observers in our system. The resulting rule is shown in Fig. 3.7. The observer associated to the Part simply updates its *optimal cycle time* by adding the time the Conveyor takes when transporting the Part.

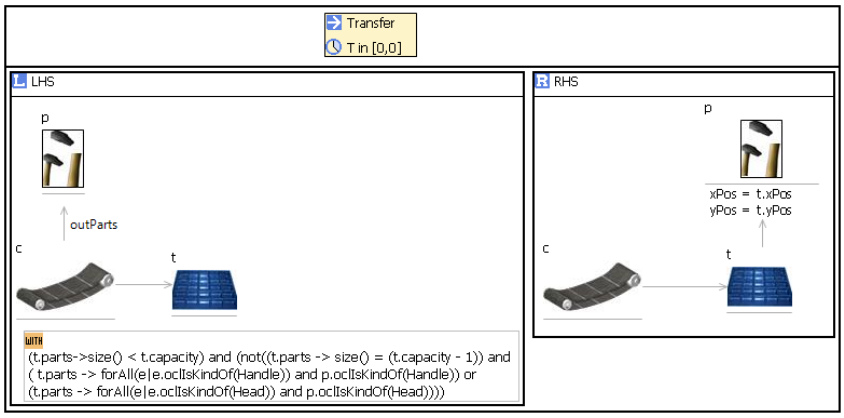


Figure 3.8: Transfer Rule.

Once there is a Part on a Conveyor ready to be transferred to the Tray connected to the end of it, the instantaneous rule Transfer (Fig. 3.8) deals with it. The LHS pattern specifies that this Part must be placed in the outParts place of the Conveyor. The rule condition forbids the triggering of the rule when the Tray is full of Parts, or when there is only one Part needed to reach the capacity of the Tray and the Parts on it are of the same type as Part p. This last condition, together with the restriction of Generators to produce a Part when their out Conveyor is full, prevents the system from deadlocking: if all the Parts in the Assembler's input Tray were of the same type, the Assembler could not assemble anymore and the production line would stop.

The Assemble rule is shown in Fig. 3.9. This rule models the behavior of generating a Hammer using one Head and one Handle. We can see in the rule's LHS that the Tray which is connected to the Assembler has to contain a Head and a Handle. The NAC indicates that this rule cannot be triggered if the Assembler is already participating in an action of type Assemble, i.e., if it is already assembling a Hammer—maybe with different Parts. In the RHS pattern we see how the Head, the Handle and their associated observers have been removed and a new Hammer has been created in the position of the Conveyor connected to the Assembler. The defective attribute of the new collected Hammer will be true either if one of the assembled Parts (or both) was defective or if the random variable (rdm) is smaller than the defective rate of the Assembler. The Hammer has an associated TimeStampOb observer whose ts attribute is the lowest value between the timestamps of the Head and the Handle. Its oct (optimal

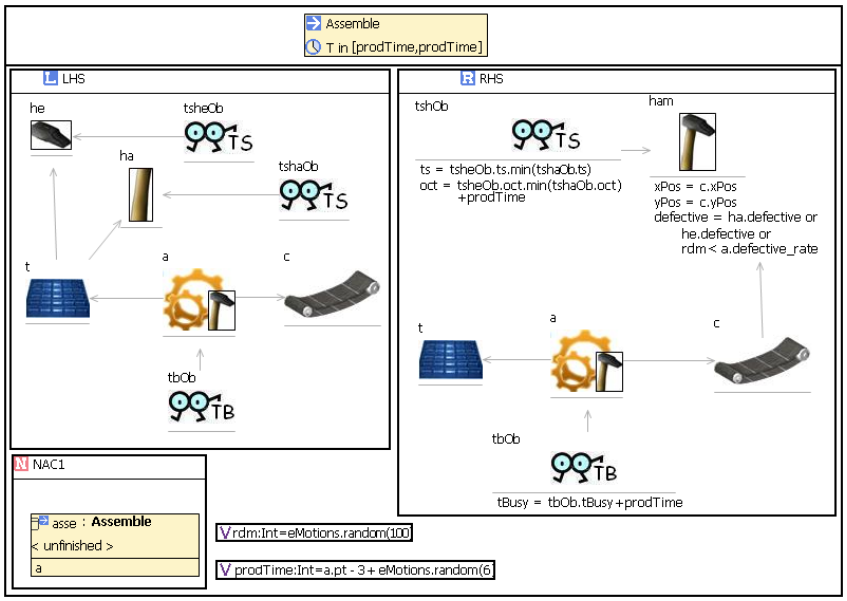


Figure 3.9: Assemble Rule.

cycle time) attribute is the lowest oct value of the Head and the Handle plus the time the Assembler has spent in assembling the Hammer, which is specified by the variable prodTime.

Fig. 3.10 shows the Collect rule. This rule models the behavior of the system when the User finally collects an assembled Hammer. The User acquires the position of the Tray where the Hammer is. The Hammer and its associated observer disappear from the system, modeling that the User has collected the Hammer, since they are no longer needed. The time this rule takes is defined by the Manhattan distance between the Hammer and the User plus one, which is stored in the variable collectTime. It models the time the User spends in getting to the Tray where the Hammer is. There is also a NAC in this rule, which forbids users to collect more than one Hammer at a time. We also see the presence of four general observers, whose states are updated when the rule is executed.

We showed in rules in Figs. 3.6 and 3.9 how TimeBusyOb observers update their tBusy attributes. Their tIdle attributes are kept up to date with an ongoing rule. This rule is shown in Fig. 3.11. It simply calculates the time the Machine associated with this observer has been idle by subtracting the time the Machine has been working from the current time elapse of the system. That same rule is used to keep the state of the ThroughPutOb and MTBOb observers updated (Fig. 3.11). The use of an ongoing rule allows this observers to keep updated at all times.

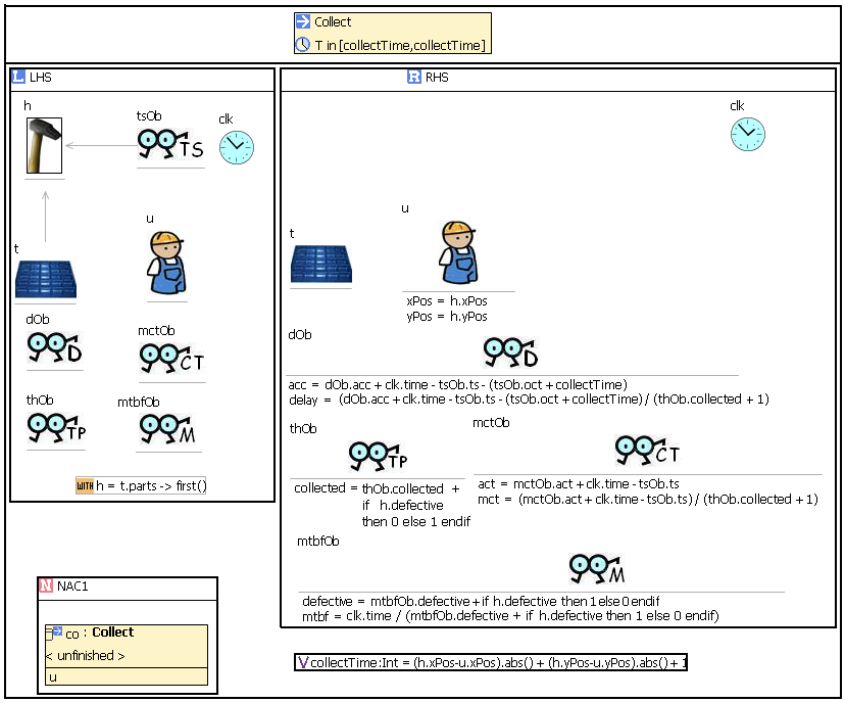


Figure 3.10: Collect Rule.

3.3 Performance Analysis of the System

The rules described in the previous sections allow users to specify the behavior of their systems and of their monitoring observers.

Once these specifications are completed, we show in this section how we can analyze them, together with the tool available for conducting such analyses.

In *e-Motions*, the semantics of real-time specifications is defined by means of transformations to another domain with well-defined semantics, namely Real-Time Maude [ÖM07]. The *e-Motions* environment not only provides an editor for writing the visual specifications, but also implements their automatic transformation (using ATL [JABK08b]) into the corresponding formal specifications in Maude—in a way transparent to the user.

One of the goals of such formal specifications is to provide formal semantics to the visual specifications of the system. More precisely, what we get when we translate the rules into Maude is a rewriting logic specification of the system. But more importantly, this approach enables the use of Maude's facilities and tools available for executing and analyzing the system specifications once they are expressed in Maude. The work in [RVD09, RDV10] present some examples of analyses that can be performed on rule-based DSL specifications using Maude. Furthermore, Maude rewriting logic specifications are executable, and therefore they can be used as a prototype of the system on which to carry on different types

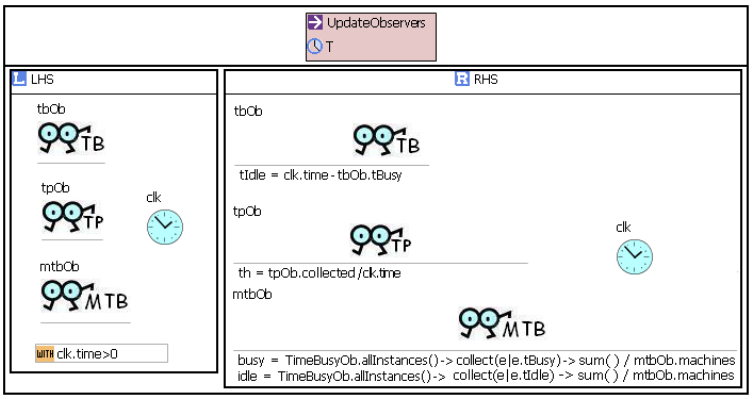


Figure 3.11: UpdateObservers Rule.

of checks and perform simulations.

In Maude, the result of a simulation is the final configuration of objects reached after completing the rewriting steps, which is nothing but a model. The resulting model can then be transformed back into its corresponding EMF notation, allowing the end-user to manipulate it from the Eclipse platform. The semantic mapping as well as the transformation process back and forth between the e-Motions and Real-Time Maude specifications are described in detail in [RDV10]. These transformations are completely transparent to the *e-Motions* user. In this way the user feels like working only within the *e-Motions* visual environment, without the need to understand any other formalisms and being completely unaware of the Maude rewriting engine performing the simulation (Figure 3.12).

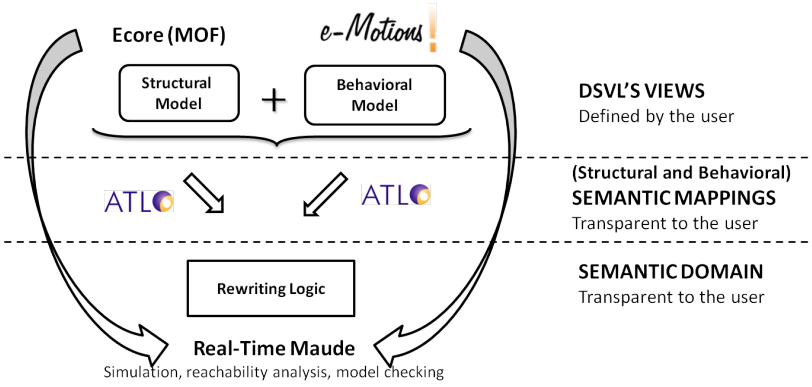


Figure 3.12: Semantic mapping between *e-Motions* and Maude.

Another very important advantage with our approach is that observers are also objects of the system, and therefore the values of their attributes can be effectively used to know how the system behaved after the simulation is carried out. For example, if we start the simulation from the initial configuration of the system

Table 3.1: Observers results after a simulation with one Assembler (pt = 40 and defective_rate = 1), one GenHandle (pt = 30 and defective_rate = 5) and one GenHead (pt = 40 and defective_rate = 2). System production time: 27'16".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.39	mtbf:9'05"	mct:4'05"	delay:2'26"	tBusy:71%	tBusy:95.4%	tBusy:95.4%
collected:38	defective:2	act:163'15"	acc:97'10"	tIdle:29%	tIdle:4.6%	tIdle:4.6%

shown in Fig. 3.5, the values obtained by the observers after running a simulation are shown in Table 3.1 (recall that the counter attributes of the generator machines were set to 40). Let us remind the reader that a given production time of 40 time units for a Machine indicates that it will take an amount of time within the range [37,43]. Similarly, a production time of 30 means that the real production time taken by that Machine to produce a Part will be within the range [27,33]. In the following tables we will consider that units of time correspond to seconds, to show the simulation results in a clearer way—e.g., we will write 9'05" (meaning 9 minutes and 5 seconds) instead of writing 545 units of time.

Table 3.1 shows the performance parameters achieved by the system, whose simulation indicated that the production time was 27'16" (this is the time the system took to produce all the Parts). Starting with the ThroughputOb observer, it indicates that 38 non-defective Hammers have been collected. The throughput value is 1.39 (expressed in parts/minute), which means that 1.39 non-defective Hammers are collected on average every minute. Two defective Hammers have been produced, and the mean time between their collection was 9'05". Depending on the price of materials, this could be of greater or lesser concern. The mean time that a Hammer is in the system is 4'05". Let us clarify that this is the time elapse from when its Parts start being generated to the moment it is collected. The average delay of all Hammers produced is 2'26", which means that they are in the system for around 2'26" more than their optimal (theoretical) production time. Finally, let us have a look at the time the Machines have been working. The head generator and assembler have been working almost the whole time (95.4%). However, the handle generator is idle one third of the time. As expected, this is because its production time is faster than the other Machines, so it produces the Parts faster and has to wait for the other Machines before being able to carry on.

We also need to study the reasons for the delay. In the first place, Handles are generated faster than Heads. This may cause an overload of Handles in Tray t1 (see Fig. 3.5) and also in Conveyor c1, which makes the handle generator stop generating Handles (see Sec. 3.2.3). This overload means that Parts stay in the system longer, increasing their cycle time and, consequently, their delay. To try to solve this problem, let us see what happens if the production time of the heads generator is set to 30" (and the defective rate to 5%) so that now Handles do not have to wait for Heads to be assembled because the production time of both

Table 3.2: Observers results after a simulation with one Assembler (pt = 40 and defective_rate = 1), one GenHandle (pt = 30 and defective_rate = 5) and one GenHead (pt = 30 and defective_rate = 5). System production time: 26'57".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.41	mtbf:8'59"	mct:3'42"	delay:2'04"	tBusy:72.4%	tBusy:73.6%	tBusy:96.3%
collected:38	defective:2	act:148'3"	acc:82'28"	tIdle:27.6%	tIdle:26.4%	tIdle:3.7%

Table 3.3: Observers results after a simulation with one Assembler (pt = 30 and defective_rate = 3), one GenHandle (pt = 30 and defective_rate = 5) and one GenHead (pt = 30 and defective_rate = 5). System production time: 20'48".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.8	mtbf:7'35"	mct:1'37"	delay:8"	tBusy:95%	tBusy:94.8%	tBusy:93.3%
collected:38	defective:2	act:64'35"	acc:5'30"	tIdle:5%	tIdle:5.2%	tIdle:6.7%

Machines is similar. The results of this simulation are shown in Table 3.2.

These results are similar to those in Table 3.1. Once again, 38 non-defective Hammers and two defective Hammers have been collected. The mean cycle time and delay of collected Hammers have been slightly reduced, but they are still significant. The explanation for this is simple. Despite handle and head generators having similar production times, now Parts which are ready to be assembled on Tray t1 have to wait for the Assembler to be available, since its production time is greater than the production time of the Generators. This is also the reason why the busy time of Generators is smaller than the Assembler's, which is not desirable. A solution to the problem of these big mean cycle times, delays and idle times of Machines seems to be making the production time of all Machines equal. As we have been simulating with two different production times (and, consequently, defective rates), let us do a simulation with all times set to 30" (and a defective rate of 5% for Generators and 3% for Assemblers, see Table 3.3), and another with 40" (and a defective rate of 2% for Generators and 1% for Assemblers, see Table 3.4).

Now we can see improvements in these two simulations regarding the mean cycle time, delay and idle time of the Machines; the mean cycle time of collected Hammers has been substantially decreased and the delay is now quite small. The reason is that all Machines work now at the same speed. The mean cycle time is bigger when the production time of Machines is 40" because Parts are within the system for a longer time until the User collects them. The busy and idle times in both situations are very similar (and acceptable), since the Machines are working for almost the whole time. As expected, the simulation when the production times are 30" finishes earlier. In fact, it now takes seven minutes less: the system produces all Parts in 20'48". Besides, the throughput of the former, 1.8, is better

Table 3.4: Observers results after a simulation with one Assembler (pt = 40 and defective_rate = 1), one GenHandle (pt = 40 and defective_rate = 2) and one GenHead (pt = 40 and defective_rate = 2). System production time: 27'52".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.4	mtbf:13'56"	mct:2'03"	delay:13"	tBusy:94%	tBusy:93.7%	tBusy:94.9%
collected:39	defective:1	act:81'53"	acc:9'01"	tIdle:6%	tIdle:6.3%	tIdle:5.1%

than the one of the latter, 1.4. However, in the former two defective Hammers have been collected, while in the latter only one Hammer was defective. Taking into consideration all these factors, we, acting as managers of the plant, should have to decide which configuration is better for our plant. For instance, if the cost of the materials were very expensive, we may prefer the second configuration, where the process takes 7' longer but we only get one defective Hammer. However, in this case, since we are dealing with Hammers, we would probably prefer to get two defective Hammers and save 7 minutes. In any case, we now count on precise performance figures that allow us to make decisions and to assess their associated costs and impact (in terms of time and money).

3.3.1 Adding Rules for Self-Adaptation

Apart from computing the values of the properties that we want to analyze in the system, observers can also be very useful for defining alternative behaviors of the system, depending on specific threshold levels. For instance, the system can self-adapt under certain conditions, since we are able to search for states of the system in which some attributes of the observers take certain values, or go above or below some limits.

As an example, let us consider the mean cycle time value given by the MCTOb observer. This value computes, each time a new Hammer is collected, the mean cycle time of all Hammers produced so far, and it directly depends on the production time of the Machines (as we saw in the previous section). Let us suppose that the production time of the Machines can be changed during execution time. In fact, it is very common in real world systems, where the working speed of different Machines can be adjusted according to certain parameters while the system operates.

Let us consider the two configurations that gave us better results in the previous simulations (those where all the Machines had the same production time). The simulation of the configuration on which every Machine has a production time of 40" (resp. 30") resulted in a final mean cycle time of 2'03" (resp. 1'37"). Now, suppose that we want to keep the value of the mean cycle time close to a given optimal value for us (let us say 1'50") by appropriately swapping the two configurations. Taking this into account, we include two new rules in the system to self-adapt its configuration (Fig. 3.6.5). The first rule, DecreasePt, will be

(a) DecreasePt Rule (b) IncreasePt Rule

Figure 3.13: The rules that describe how the system configuration self-adapts.

Table 3.5: Observers results of simulation with the system being self-adapting and pursuing a mct of 1'50". System production time: 22'59".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.7	mtbf:11'30"	mct:1'50.3"	delay:15"	tBusy:95%	tBusy:94.9%	tBusy:93.5%
collected:39	defective:1	act:73'33"	acc:10'10"	tIdle:5%	tIdle:5.1%	tIdle:6.5%

triggered whenever the current mean cycle time of the system goes above 1'50" and the current production time of the three Machines is 40". This rule will change the production times of every Machine to 30" (and consequently their defective rate) with the aim of raising up the mean cycle time. Rule IncreasePt will then be triggered whenever the mean cycle time goes below 1'50" and the production time of the three Machines is 30". It will change the production times of every Machine to 40" (and accordingly their defective rate) to decrease the mean cycle time.

For checking the efficiency of this approach, we have carried out a simulation where the initial production time of every Machine is 40". The result of this simulation is presented in Table 3.5, which shows how the mean cycle time is close to 1'50", as desired. In fact, we see that this is the most appropriate configuration: we obtain a throughput very close to the highest before (1.8) but with only one defective Hammer.

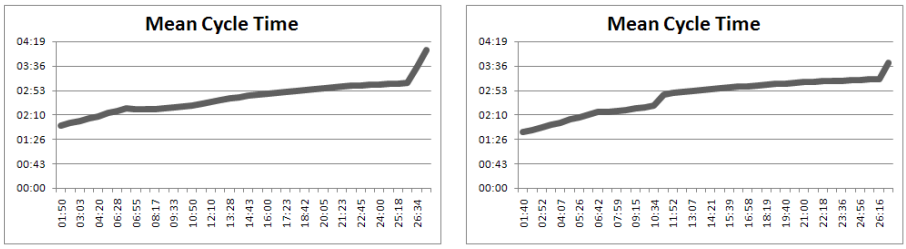
3.3.2 Storing Attributes in Traces and Exporting Analysis Results

So far we have seen how it is possible to simulate the system and obtain performance information on its behavior. But there are also occasions in which we are not only interested in the performance indicators at the end of the system simulation, but also during its execution. For instance, we are now able to know the average cycle time for the produced Parts, but it might be useful to see a graphical representation of how said value is changing while the system is operating. In addition, it would be important to have the resulting models designed in a way which can be analyzed and displayed by different tools (math programs, spreadsheets, etc.), outside the EMF environment.

This section shows how to achieve this and the tool support that *e-Motions* provides. In the first place, if we want to keep track of how the values of the observed properties change during the system execution it is just a matter of changing the observers' attributes to be sequences of values. And then the behavior of the observers needs to be slightly modified in the rules to append every computed value to these sequences (instead of gathering only the final value). This can be useful when we want to analyze the values of some property along time. It could also be useful if the modeler wants to use the values of the traces within the rules. For example, the Batch Means function described later in Section 3.5.2 uses these values.

Regarding the use of the resulting models by other tools, we have added to *e-Motions* implementation a trivial model-to-text transformation that enables the creation of a comma-separated values (*csv*) file from an Ecore model. Such a *csv* file contains the information of every object in the model, together with the values of all its attributes. Objects are named by their identifiers, and attributes are expressed as a list of name-value pairs. That file can be directly imported by different applications for performing different kinds of analysis. For example, it can be fed to an spreadsheet application that the domain expert can use to analyze the data, display charts, etc. In this way, the domain expert will be able to easily display charts with the result of a simulation (which is in fact a model) to graphically represent the values of the parameters monitored by the observers throughout the whole simulation.

Fig. 3.14 shows, for instance, two charts that display the mean cycle time of the simulations whose final results were shown in Table 3.1 and Table 3.2, respectively. We can see in both of them how, in general, the mean cycle time of Parts grows as time goes by. This is because Tray t1 gets overloaded at some point in the simulation, and when this occurs the generated Parts at that moment increase the mean cycle time. However, in some time intervals the mean cycle time does not vary or even slightly decreases. This is caused by the Parts that are produced when the Generators re-start their work after stopping to avoid a deadlock (as discussed in Sec. 3.2.3), since these new Parts will have a smaller mean cycle time than the average.



(a) Production times 40-30-40 (b) Production times 40-30-30

Figure 3.14: Mean Cycle Times.

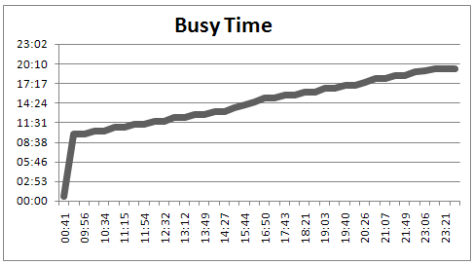


Figure 3.15: Handles generator busy time 40-30-40.

Fig. 3.15 shows the busy time of the handle generator in the simulation whose final results were shown in Table 3.1. In this simulation, the production time of the handle generator was smaller than the other Machines. Therefore, Handles were generated faster than Heads and the handle generator had to eventually stop generating Handles to avoid the overload of Parts in Tray t1 and the overload of Heads in Conveyor c2. In particular, we see that the Generator stops for the first time around minute 8 (moment at which the overload happened) and after this moment it continuously restarts and stops the generation of Parts as needed, working as expected.

This analysis also indicates further potential points for improvement in the system, such as replacing the current Conveyors with faster ones (although probably more expensive—hence another trade-off to consider). Finding the right balance between costs, performance and benefits is not easy, although the kinds of analysis presented here can help solve this problem at the earliest phases of the system's design, with precise and objective figures, and also using notations and mechanisms which are quite close to the domain experts.

3.4 Adding Probability Distributions

In the previous section we have described how the random(n) function can be used to assign an interval to the duration of a rule. For example, the duration of the

Assemble rule was in the range $[a.pt-3, a.pt+3]$. What we are in fact modeling is a uniform probability distribution with the form $unif(a.pt-3, a.pt+3)$.

Normally the behavior of a non-trivial system is stochastic, and therefore it needs to be defined using probability distributions that specify the rates at which external events occur (e.g. arrival rates), the duration of the rules, the probability of failures, etc. Once we count on a random number generator, which actually models a uniform distribution, we can implement other random number generators that follow different probabilistic distributions. For example, having a uniform random number, u, in $[0,1)$, we can calculate $exp(\lambda) = \frac{\ln(1-u)}{-\lambda}$, which is a random number generator with an exponential distribution and rate parameter λ. That formula is called the *inversion method*.

What the random generator available in *e-Motions* concretely uses internally is the Mersenne twister random number generator that Maude provides [CDE+07]. Since the models and rules specified in *e-Motions* are then translated to Maude, the auxiliary functions that we can use are implemented in Maude. This is, we have developed a Maude module where the previous formula for exponential random numbers is implemented. The whole set of probabilistic distributions, as well as their formulae and way of calling them, is the following:

- *Exponential* distribution (Fig. 3.16). It describes the time between events in a *Poisson* process, i.e., a process in which events occur continuously and independently at a constant average rate. In this way, this distribution is memoryless. To obtain a value exponentially distributed in *e-Motions*, eMotions.expDistr(λ) must be typed, where λ>0 is the rate of the distribution. λ can be either an integer or a double value. When defining a variable (duration) to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.expDistr(0.5). The formula that implements this function is the previously presented inversion method.

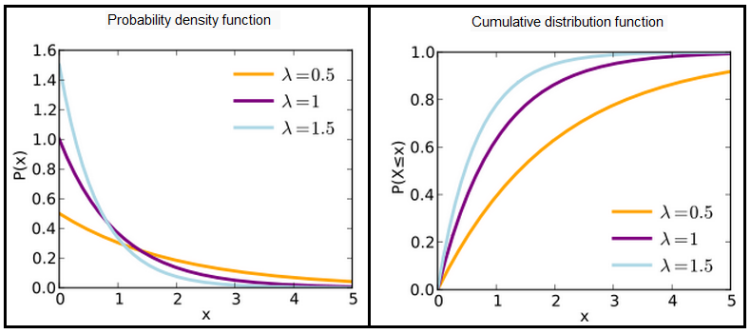


Figure 3.16: *Exponential* distribution.

- *Normal* distribution (Fig. 3.17). The *Normal* (or *Gaussian*) distribution is a continuous probability distribution that is often used as a first approxima-

tion to describe real-valued random variables that tend to cluster around a single mean value. To obtain a value normally distributed in *e-Motions*, eMotions.normDistr(μ,σ) must be introduced, where μ is the mean and σ is the square root of the variance. μ and σ can be either integer or double values. When defining a variable (duration) to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.normDistr(0.7,2). The formula we use to implement this function is called the *Box-Muller* method, which uses two different random number generators. Such method is: $norm(\mu,\sigma) = \sigma \times norm(0,1) + \mu$, where $norm(0,1)$ is the standard normal distribution and is calculated as $norm(0,1) = \sqrt{-2\ln(rnd(0,1))} \times \cos(2\pi \times rnd(0,1))$.

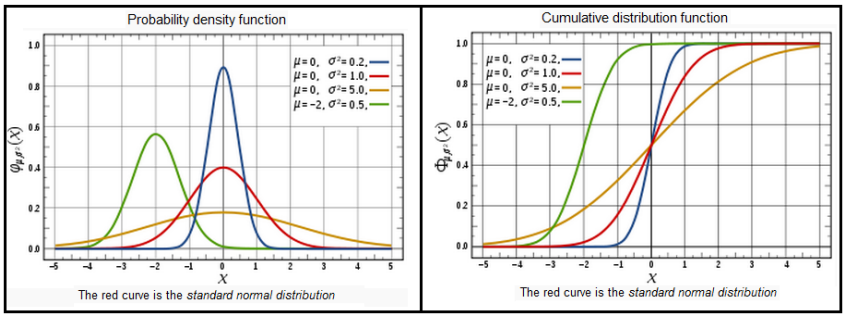


Figure 3.17: *Normal* distribution.

- *Erlang* distribution. The Erlang distribution is a continuous probability distribution with wide applicability primarily due to its relation to the *Exponential* and *Gamma* distributions. In fact, it is like the *Gamma* distribution with the restriction that the shape must be a positive integer, and is an extension of the *Exponential* distribution. Nowadays, it is used in the fields of stochastic processes and of biomathematics. To obtain a value distributed according to this distribution in *e-Motions*, erlangDistr(s,k) must be typed, where s>0 is the scale and k>0 is the shape. k is an integer value and s can be either an integer or a double value. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.erlangDistr(2.2,1). We have implemented it with the following formula, which reuses the formula of the inversion method showed before: $erl(s,k) = \sum_{i=1}^{s} exp(k)$.

- *Gamma* distribution (Fig. 3.18). This distribution is frequently a probability model for waiting times; for instance, in life testing, the waiting time until death is a random variable that is frequently modeled with a gamma distribution. It is a generalization of the *Erlang* distribution and allows non-integer shape parameters. To obtain a value distributed according to this distribution in *e-Motions*, eMotions.gammaDistr(s,k) must be used, where

k>0 is the shape and s>0 is the scale. k and s can be either integer or double values. When defining a variable (duration) to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.gammaDistr(2,0.6). The formula used to implement it is the same as the *Erlang*'s one, but now it considers non-integer shape parameters.

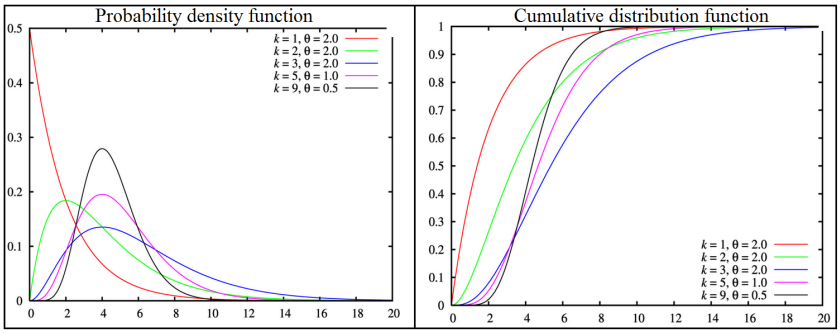


Figure 3.18: *Gamma* distribution.

- *Weibull* distribution (Fig. 3.19). It is a versatile distribution that can take on the characteristics of other types of distributions, based on the value of the shape parameter. It is one of the most widely used lifetime distributions in reliability engineering. In fact, the *Weibull* distribution is a very flexible life distribution model that can be used to characterize failure distributions in all three phases of the bathtub curve [KKW03]. To obtain a value distributed according to this distribution in *e-Motions*, eMotions.weibDistr(λ,k) must be typed, where λ>0 is the scale and k>0 is the shape. λ and k can be both integer and double values. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.weibDistr(0.2,2). In order to simulate the *Weibull* distribution, the following formula has been used: $weib(\lambda,k) = (\frac{-1}{\lambda})(\ln(1-rnd(0,1)))^{1/k}$.

- *Chi-square* distribution (Fig. 3.20). This distribution, with k degrees of freedom, is the distribution of a sum of the squares of k independent standard normal random variables. It is one of the most widely used probability distributions in inferential statistics, e.g., in hypothesis testing or in construction of confidence intervals. This distribution is a special case of the *Gamma* distribution. To obtain a value distributed according to this distribution in *e-Motions*, chiSDistr(k) must be typed, where k>0 represents the degrees of freedom. k can be either an integer or a double value. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: $duration : Int = eMotions.chiDistr(150)$. Being

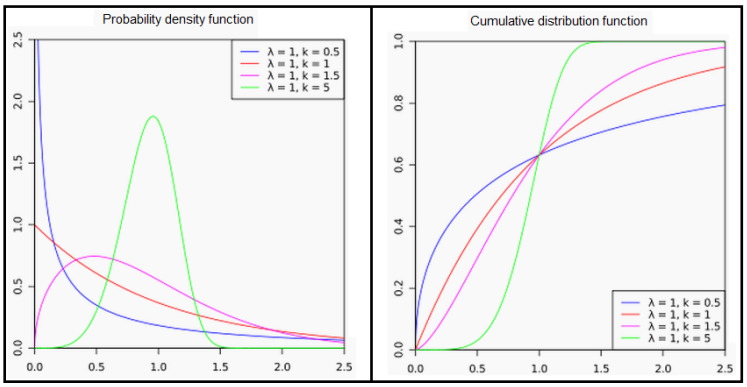


Figure 3.19: *Weibull* distribution.

the *Chi-square* distribution a special case of the *Gamma* distribution, we get *Chi-square* random values with the following formula, which uses the shape parametrization of the *Gamma* distribution: $chi(k) = gamma(\frac{k}{2}, 2)$.

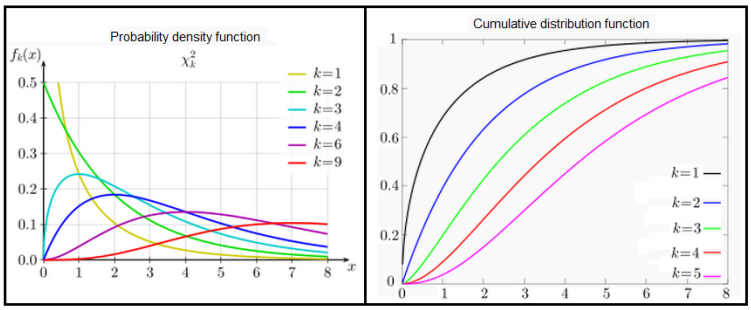


Figure 3.20: *Chi-square* distribution.

- *F* distribution (Fig. 3.21). The ratio of two *Chi-square* variables has an *F* distribution. It is also known as *Snedecor's F* distribution or the *Fisher-Snedecor* distribution. The F distribution has two degrees of freedom, d1 for the numerator, d2 for the denominator. For each combination of these degrees of freedom there is a different *F* distribution. It is more spread out when the degrees of freedom are small. As the degrees of freedom increase, the *F* distribution is less dispersed. To obtain a value distributed according to this distribution in *e-Motions*, fDistr(d1,d2) must be introduced. d1 and d2 can be either integer or double values. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.fDistr(110.5,57.3). For the generation of random numbers following an *F* distribution, we use the *Chi-square* distribution: $f(d1, d2) = \frac{chi(d1)/d1}{chi(d2)/d2}$.

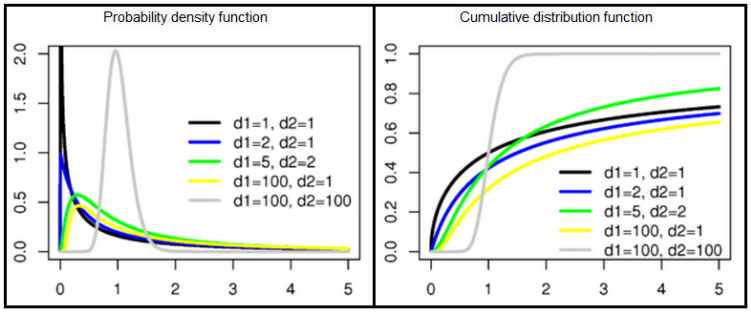


Figure 3.21: F distribution.

- *Geometric* distribution (Fig. 3.22). The distribution of number of trials up to and including the first success in a sequence of *Bernoulli* trials is called a *Geometric* distribution. It is a discrete equivalent of the exponential distribution, so it is memoryless. To obtain a value distributed according to this distribution in *e-Motions*, geomDistr(p) must be introduced, where p is the mean. p can be either an integer or a double value. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.geomDistr(2.6). The formula used to generate random numbers geometrically distributed is basically the same used in the *Exponential* distribution: $geom(\lambda) = \lfloor \frac{\ln(1-u)}{-\lambda} \rfloor$.

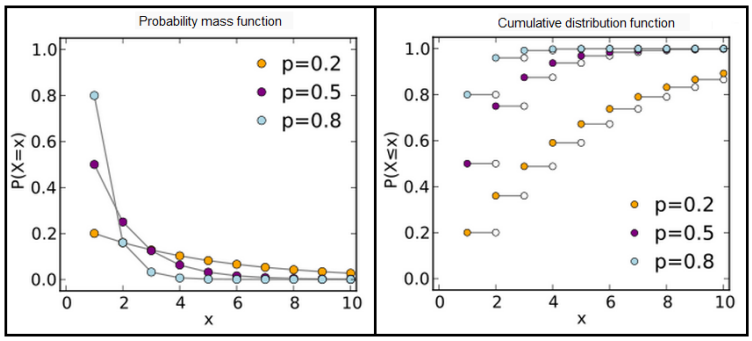


Figure 3.22: *Geometric* distribution.

- *Lognormal* distribution (Fig. 3.23). The logarithm of a normal variate has a *Lognormal* distribution. In regression modeling and analysis of experimental designs, often log transformation is used. In such cases, the response in the transformed model has a *Normal* distribution while the original response has a *Lognormal* distribution. The *Lognormal* distribution describes many naturally occurring populations. In the mining and extraction industries, it has been observed that where the value of an item is proportional to size, the

population is probably lognormally distributed, with a few valuable items and a lot of uncommercial items. To obtain a value distributed according to this distribution in *e-Motions*, logNormDistr(μ,σ^2) must be introduced, where μ is the log-shape and σ^2 is the scale. μ and σ^2 can be both integer and double values. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.logNormDistr(200.5,10). In order to generate lognormally distributed random numbers, we use the formula of the *Normal* and *Exponential* distributions: $lognorm(\mu,\sigma^2) = exp(\frac{1}{norm(\mu,\sigma^2)})$.

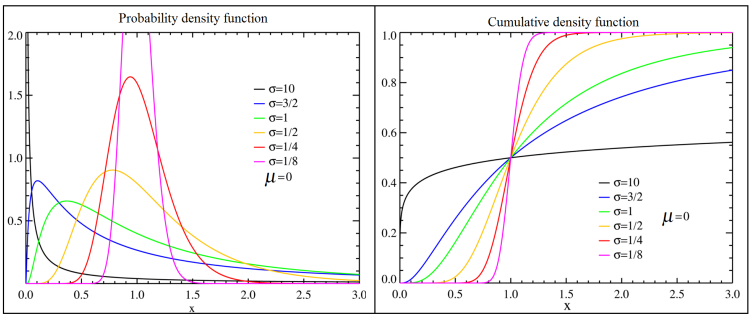


Figure 3.23: *Lognormal* distribution.

- *Pareto* distribution (Fig. 3.24). It is a power law probability distribution that coincides with social, scientific, geophysical, actuarial, and many other types of observable phenomena. Outside the field of economics, it is sometimes referred to as the *Bradford* distribution. To obtain a value distributed according to this distribution in *e-Motions*, eMotions.paretDistr(s,k) must be introduced, where k>0 is the shape and s>0 is the scale. k and s can be both integer or double values. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.parentDistr(0.5,10). Random samples can be generated using inverse transformation sampling: $par(s,k) = \frac{s}{rnd(0,1]^{1/k}}$.

- *Pascal* distribution (Fig. 3.24). It is an extension of the *Geometric* distribution. In a sequence of *Bernoulli* trials, the number of trials up to and including the m_{th} success has a *Pascal* distribution. It is as well a special case of the negative binomial distribution. There is a convention among engineers, climatologists, and others to reserve "negative binomial" in a strict sense and "Pascal" for the case of an integer-valued stopping-time parameter n. To obtain a value distributed according to this distribution in *e-Motions*, eMotions.pascalDistr(n,p) must be introduced, where n>0 is the number of successes and 0<p<1 is the probability of success. n and p can be both integer or double values. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within

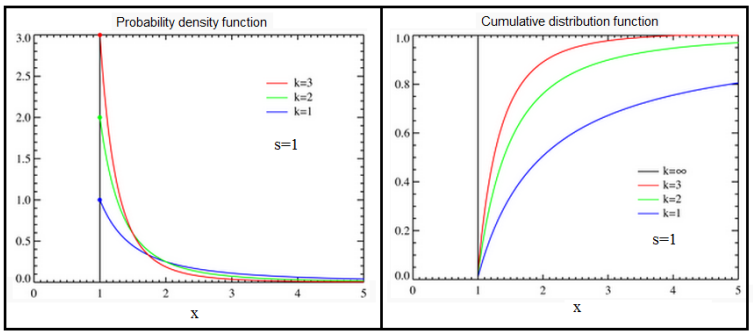


Figure 3.24: *Pareto* distribution.

the rule: duration : Int = eMotions.pascalDistr(0.5,0.7). Being this distribution an extension of the *Geometric* distribution, we can generate random numbers by generating n geometric variables $geom(p)$ and returning their sum. That would be $pasc(n, p)$.

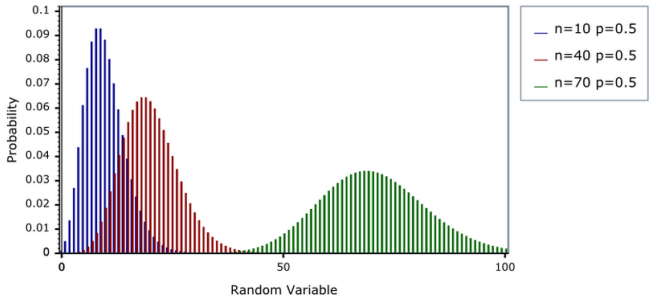


Figure 3.25: *Pascal* distribution.

- *Uniform* distribution. It is a probability distribution whereby a finite number of equally spaced values are equally likely to be observed; every one of n values has equal probability 1/n. To obtain a value distributed according to this distribution in *e-Motions*, eMotions.unifDistr(l,u) must be typed, where l is the lower bound and u is the upper bound. l and u can be both integer and double values. When defining a variable to use in the duration of a rule, here it is an example of how it should be defined within the rule: duration : Int = eMotions.unifDistr(1.5,3.7). To generate random numbers, we use the random function explained before.

In Appendix A it is shown the Maude implementation of these formulae.

3.5 General Methodology

After having described our proposal with a running example, this section presents a general methodology for the performance analysis of those systems which are specified using rule-based domain-specific languages—such as *e-Motions*. In addition, it explains the current tool support and how the simulations can be conducted with *e-Motions* to obtain the performance analysis results.

3.5.1 The Process

In order to carry out the specification of a system and to obtain performance metrics by simulating the specifications with *e-Motions*, the following steps are required:

1. All the relevant elements of the system have to be specified. In an MDE setting, this is done by defining the abstract syntax of the system, which is specified by means of its metamodel. It describes the concepts of the system and the relationships between them. The concepts of the system are defined as classes in the metamodel. Element features are specified by means of class attributes.

2. Once we know the elements that may be present in the system, their relationships and features, the system behavior needs to be defined. In our approach this is specified by means of a set of rules, each of which represents a possible action in the system. These rules should be enough to cover all the aspects of interest for the system. The form of the rules is described in Section 2.2.

3. We then need to specify the initial configuration of the system, which is nothing but a model conforming to the system metamodel. It can be defined either by a behavioral rule whose left-hand side is empty (as in our case, see Section 3.1) or creating a dynamic instance of the metamodel. In the latter approach, the initial configuration of the system can be defined using a tree view form or using a graphical editor that must be previously developed [Ecl08]. Since developing a graphical editor means more effort, we recommend to define the initial configuration by either a behavioral rule or a dynamic instance of the metamodel using the tree view form.

As a result of these steps, we obtain the specification of the basic structure and behavior of the system, in a way that can be simulated. This simulation will execute the rules starting from an initial configuration of the system, which will be changing as the simulation evolves and the rules are applied. The result of the simulation will be a model (representing the final state of the system after the simulation) which conforms to the system metamodel.

If we also want to measure the performance of the system and conduct an analysis over a set of non-functional properties, the following additional steps are required:

4. We need to identify the non-functional properties that we want to analyze in the system, such as throughput, MTBF, delay, cycle time, etc. Section 3.2.1 describes some of these properties and how they can be defined.

5. Once we know the non-functional properties to analyze, we have to define the *Observers* that will be in charge of monitoring the state of the system and its elements during the simulation. For this, a metamodel for the observers has to be defined. There can be observers monitoring properties of the global system and observers monitoring individual elements' features. Special care has to be taken when defining the attributes of the observers. Data structures must be properly defined depending on what we want to monitor. For example, for a specific feature of a specific observer we may be interested in storing just the last value, all the values taken during the simulation, or an aggregated of these (e.g., the average). To monitor the last value of an element, we only need an attribute of a simple type (Integer, Double, String...); to monitor all the values taken during the simulation we need a collection type (Sequence, OrderedSet, Set...); and for the aggregated value we normally need more than one attribute (e.g, one to keep the number of values so far, another for their accumulated sum, etc.). The metamodel for the observers used in our running example is shown in Section 3.2.2.

6. When we know and have defined the observers for our system, we need to specify their behavior, and how their values are computed as the system executes. For this, we have to modify the system behavioral rules defined in step 2 with the behavior of observers. In Chapter 6 we show another approach to perform this, based on establishing correspondences between the observers' behavioral rules and the system's rules.

   Examples of how rules are enriched with observers are discussed and shown in Section 3.2.3.

7. In the specification of self-adaptive systems, rules that make some changes in the configuration of the system depending on the observers' attributes may also be needed. An example was shown in Fig. 3.6.5, where the production time and defective rate of machines changed depending on the current value of the mean cycle time. Further changes would also be possible, such as the inclusion of new part generators, new assemblers, etc.

Once the complete behavior of the system and of the observer objects have been defined, we are in the position to run the simulations, visualize and analyze the results, and make those changes to the system that are required to improve

its performance and behavior according to our requirements. These steps are described in detail in the following sections.

3.5.2 Conducting the Simulations

Normally, the behavior of a non-trivial system is stochastic, and therefore it needs to be defined using probability distributions that specify the rates at which external events occur (e.g. arrival rates), the duration of the rules, the probability of failures, etc. The treatment of these stochastic events and the use of random data implies that two different simulations of the same system may produce different performance results, since the distributions provide different values. In order to deal with this, the user should launch several simulations for the same input model, and should aggregate the results appropriately.

There are several issues to be considered here: the length of the simulations, the number of simulations that have to be carried out to get meaningful results, and the aggregation function to use for combining the results.

For some systems, it is useful to set a time limit for the simulations, since the user may want to see the state of the system after a given number of time steps. In other cases, we are interested in simulating the system until a stable value is found (if it exists). There are different methods described in the literature to determine when the steady state of a simulation is reached, namely long runs, proper initialization, truncation, initial data deletion, moving average of independent replications, or batch means [Jai91].

In our case we have implemented both the *long runs* and the *batch means* method (with slight modifications). The long runs method is useful when simulations do not last much, and many of them can be executed rapidly. In this case, the observers do not need to keep track of all intermediate states, just the final results. The original batch means method requires running a long simulation and later dividing it up into several parts of equal duration, which are called batches. The mean of the observations within each batch is called the batch mean. The method requires studying the variance of these batch means as a function of the batch size. What we do, instead of running a long simulation and then dividing it, is to apply the method at certain points during the simulation as it moves forward. For that we store the values of the performance parameters in traces to be able to apply this method over them. In our PLS, we have applied the batch means method over the traces whenever 20 new hammers are assembled. We consider that a simulation reaches its steady state when the variance of the batch means of a trace goes below a threshold value (10^{-x}). Such value is different depending on the system and the precision that we want to get. Our results were obtained for 10^{-6}.

Finally, the results of a set of simulations need to be aggregated in order to provide one meaningful final result. Traditionally, the final value is obtained by calculating the average of the resulting values, and this is the way we currently compute our final result. However, the average does not always produce the most meaningful results. As future work we plan to analyze the results given by the

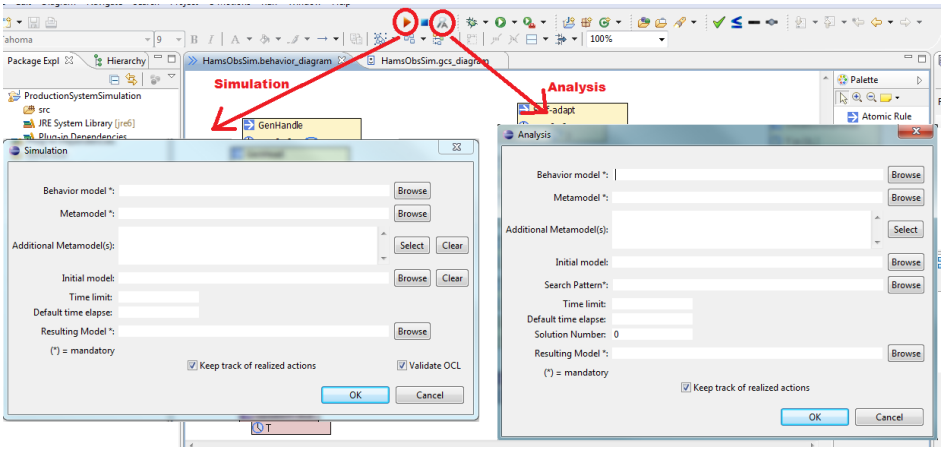


Figure 3.26: Simulating and Analyzing in *e-Motions*.

different simulations to return not the mean, but the probability distribution that the results from the different executions follow.

3.5.3 Tool Support

Our tool, *e-Motions* [Ate11], has three main functionalities: model edition, model simulation and model analysis. The first two are directly related with the work here and the third one was developed for further kinds of analysis.

e-Motions provides model editors to graphically define the abstract and concrete syntaxes of DSVLs, and their behavioral rules. An abstract syntax is defined by means of a metamodel. The concrete syntax specifies how the domain concepts included in the metamodel are represented, and it is defined as a mapping between the metamodel and a graphical notation. This graphical notation is used to define initial models and also for specifying behavioral rules.

Once the metamodel and the behavioral rules (either with observers or not) are defined, the user can simulate the specifications (see Fig. 3.26). When launching a simulation, the user indicates: the file with the metamodel; the file with the behavioral rules; the time limit for the simulation (if not specified, the simulation runs until no further rule can be triggered); the default time elapse (1, unless otherwise specified); the initial model, and the file that will store the resulting model returned by the simulation. Before carrying out the simulation, *e-Motions* applies an integrated OCL validator that checks if the OCL expressions used in the rules are correct. In case of errors, these are displayed and the user warned. Otherwise, three ATL transformations are automatically applied to transform the specifications into Maude code. One of them transforms the metamodel to its Maude specification [RVD09, RRDV07], the second one transforms the behavioral rules into Maude rules [RGdLV08], and a third one is applied if there is an input file with the initial model [RVD09].

Immediately after the transformations are applied, the simulation is launched. All this happens transparently to the user, who does not need to be aware of the existence of the Maude back-end tool, neither of the transformations happening in the background. Indeed, no Maude code is shown in the *e-Motions* tool. The simulation is executed in background, so the tool is not blocked and users can continue with their activities. A status bar is shown as long as the simulation is running. When it finishes, a notification message is shown and the file with the resulting model is created. This is a model conforming to the metamodel of the system. The user can also transform the data in the resulting model into a *csv* file by simply right-clicking on the file and selecting the transformation.

As mentioned above, random functions and probability distributions can be used in the behavioral rules. The most common distributions have been implemented and they are explained in Section 3.4.

e-Motions also offers some analysis options, for example to perform *reachability* analysis. The tool makes use of the Maude search command, which allows users to explore, following a breadth-first strategy, the reachable state space of the system in different ways. Thus, it is possible for example to look for deadlock states without needing to simulate the whole system. In our example, a possible deadlock state is that tray t1 is full of parts of the same type, preventing the assembler machine from continuing. Once again, the reachability analysis is conducted from the *e-Motions* tool (see Fig. 3.26), which, again, hides the whole interaction with Maude from/to the user. When performing reachability analysis, the user has to specify the behavior model, the metamodel, the initial model, the search pattern (for example looking for deadlocks), the maximum number of solutions to return (since it may return hundreds of solutions) and where to store the resulting models.

Finally, it is important to remark that although our proposal has been implemented on top of *e-Motions*, it is applicable to any other rule-based domain-specific language that is able to describe the behavior of the system in terms of in-place rules, and that admits the specification of time-related properties. Examples of these languages are MOMENT2 [BÖ10] or the approach presented in [dLGB$^+$10], which proposes an alternative way to incorporate time to graph transformations.

3.5.4 Pros and Cons

This section discusses some of the main advantages and limitations of our proposal concerning performance analysis of systems.

As major advantages, our approach permits obtaining the performance analysis of end-user defined DSVLs, and is supported by a prototype tool. As long as the user is capable of describing the structure of the system in terms of a metamodel and can give behavioral semantics to it by means of in-place rules, the system can be simulated and analyzed. Furthermore, many different kinds of analysis can be performed. In fact, users can define as many types of observers as needed in order to analyze the properties of interest, which provides a high degree of

flexibility. The models resulting from the simulations can be transformed into a format compatible with spreadsheet applications, so charts and tables with performance data can be easily obtained. Another advantage of our approach is that, in spite of using Maude for simulation, the user does not have to deal with any Maude code, since it is generated and used in a transparent way.

The possibility of using probability distributions in the behavioral rules enables the acquisition of performance metrics for stochastic systems, such as queuing networks (we present an implementation for queuing networks using DSLs in Chapter 5), Petri nets, stochastic neural networks and genetic algorithms.

But our approach also presents some limitations. Firstly, learning to specify systems in terms of metamodels and behavioral rules is not obvious, and the learning curve is not negligible. Secondly, we also have to consider the time required to master the tool. Finally, simulations are always expensive (time-wise). The efficiency of our current implementation can be improved in some ways. Although simulating small systems with *e-Motions* is efficient and rather fast (in the order of a few seconds), as the complexity of systems grows (in terms of number of model elements) the simulations become slow. The use of DSVLs normally implies that models are not very large, because of the high-abstraction level at which the system is specified. However, there are cases of models in which the high number of elements (e.g., greater than 200 at this time) represents a heavy burden for *e-Motions*. In next chapter, we slightly modify our approach to reduce the number of objects in our models by modeling the elements that flow within the system as attributes instead of classes. Furthermore, we are studying alternative representations of the models and behavioral rules in Maude to improve the speed of the simulations, as well as the parallel distribution of simulations among servers. In this way, a simulation where probability distributions take part would be distributed over several machines, each one dealing with one system simulation. When the simulations are finished, the resulting models are gathered and the corresponding results are returned.

Table 3.6: Simulation times for the Production Line system.

# Model Elements	With no Observers	Global Observers	Individual Observers
30	0:00:01	0:00:01	0:00:12
100	0:00:06	0:00:18	0:02:00
120	0:00:12	0:00:34	0:04:26
150	0:00:35	0:01:07	0:08:51
200	0:01:12	0:01:56	0:17:43
300	0:02:28	0:03:32	0:59:42
400	0:03:45	0:05:04	1:59:24
500	0:05:51	0:07:38	4:56:07
600	0:07:31	0:10:30	8:36:08
750	0:11:16	0:17:15	14:51:18
900	0:15:33	0:32:12	31:05:04
1000	0:19:55	0:36:45	127:26:48

We have also evaluated the overhead of introducing the observers in the system specifications. They are objects, too, and hence they may mean multiplying by two or three the number of objects in the model if individual observers are used for all system elements. For small systems this is an acceptable increase: for systems with less than 100 elements the simulation time goes from a few seconds to a few minutes. However, the exponential nature of the Maude simulations and our current implementation of the tool impose a limit on the number of elements if simulations are to be finished in a reasonable time—see Table 3.6.

3.6 Packet Switching Case Study

In this section we present the case study in [TBLRV11]. We show it in order not to limit our approach to a concrete case study. This way, we prove that we can apply it to different kinds of systems, even with some complexity despite the high level of abstraction of the approach. Furthermore, we show that many different performance metrics can be obtained, and we follow the steps described in Section 3.5.1.

3.6.1 System to model

Our aim is to model a re-configurable communication network system, composed of computers that transmit messages through nodes that process and forward them to other nodes until messages reach their final destinations. Additional supporting nodes can be activated in case of network congestion to alleviate the temporary traffic bottlenecks. Assuming that the cost of acquiring and maintaining these extra nodes is not negligible, there are some tradeoffs between the quality of service provided by the network and its overall cost. We show how this kind of analysis and tests can be conducted with our proposal in an easy and cost-effective manner.

3.6.2 Steps 1, 2 and 3

These steps involve the definition of the abstract syntax, behavior and initial configuration of the system. The metamodel of the network is shown in Fig. 3.27. It is a communication network composed of different kinds of Components that can contain Packets. Each Component has a specific location, given by two coordinates (xPos and yPos).

Users produce a number of Packets given by their counter attribute, while Servers consume them (i.e., they act are sources and sinks of the network, respectively). Components can exchange Packets only if they are connected. Such connection between components is modeled by the neighbours reference, that reflects the Components that are reachable from a given Component. The network itself is modeled by a set of packet switching Nodes, which are the network

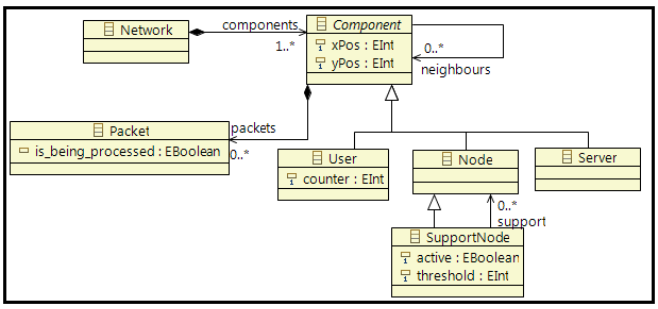


Figure 3.27: Communication Network Metamodel.

elements in charge of receiving, processing and forwarding Packets to other Components. The buffer with the set of received Packets that a Node has to process is modeled by means of the packets relationship between Component and Packet.

One characteristic of this network is that the time each node spends in processing a Packet depends on the number of Packets in its buffer. The more Packets in the buffer, the longer the Node takes to process each one. This simulates a behavior where Nodes need to perform some operations on the flow of Packets, such as sorting or merging them according to a given algorithm, for instance. Packets have an attribute, is_being_processed, that indicates whether the packet is currently being processed by a node. For Packets' routing, Nodes decide to forward them to the neighbor Node which is less loaded, i.e., the one with the smallest number of Packets in its buffer.

In order to alleviate network congestion, an additional kind of nodes (called SupportNode) exists in the network. They can be activated and de-activated depending on the load of their neighboring nodes. Each SupportNode activates itself if the number of Packets in the buffer of any of the Nodes connected to it via the support relationship goes above the value defined in its threshold attribute. Similarly, it deactivates itself when the load of all connected nodes is below the threshold.

The behavior of the network is given by a set of in-place behavioral rules. For example, rule Forwarding (Fig. 3.28) models the forwarding of Packets among Components and Nodes. This rule is fired when sending Packets from Users to Nodes and from Nodes to Nodes. To apply this rule, the Packet must not be being processed. Furthermore, there are two OCL constraints that have to be satisfied in order to launch this rule. They state that the target Node is the Component's neighbour which is processing the lowest number of Packets, and that it cannot be a deactivated SupportNode. In the rule's RHS, the Packet has moved to the Node and it has started been processed. The duration of this rule can be either 0 or 1 time units (the fact that a Packet can take 0 units simulates the situation in which several Packets are forwarded together to optimize an open connection). The remaining rules are shown later together with the observers' behavior.

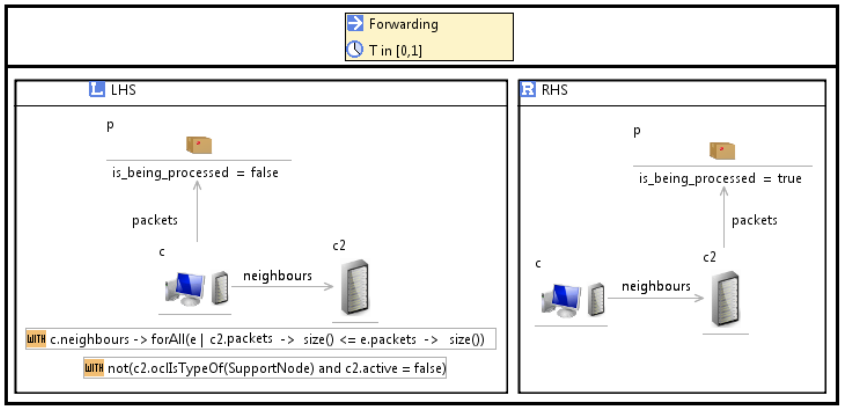


Figure 3.28: Forwarding Rule.

The initial configuration of a network is nothing but a model that conforms to the Network metamodel. A possible configuration is shown in Fig. 3.29 (please ignore the area within the dotted lines for now). We can see the concrete visual syntax in such figure. This configuration defines four Users feeding Packets into the network (each one will introduce 125 Packets) and one Server consuming them. Each User accesses the network using different Nodes. The network is composed of 8 (normal) Nodes and 2 SupportNodes (n9 and n10), which are initially inactivate. Their activation depends on the buffer size of Nodes n3 and n4 for SupportNode n9, and of Nodes n5 and n6 for SupportNode n10. This is specified by the corresponding support relationships between the SupportNodes and the Nodes they try to clear.

3.6.3 Step 4

This step identifies the QoS properties to analyze. Let us assume that the cost of acquiring, maintaining and running these extra support nodes cannot be ignored, as it happens for instance if support nodes are hired from external network providers, and their running costs depend on the time they are active or on the number of packets they process. In this setting, the system owner is faced with several decisions in order to maximize the quality of service provided by the network while minimizing its overall cost. Firstly, how many supporting nodes need to be hired/purchased to guarantee a minimum level of throughput? Secondly, which is the optimal value for the threshold of each support node that provides a required level of throughput with the minimum time of support node activation (hence minimizing the running cost of the node)?

In order to be able to respond to these questions, we need to identify which are the system parameters that are relevant to our analysis. In our case, we will focus on the following ones:

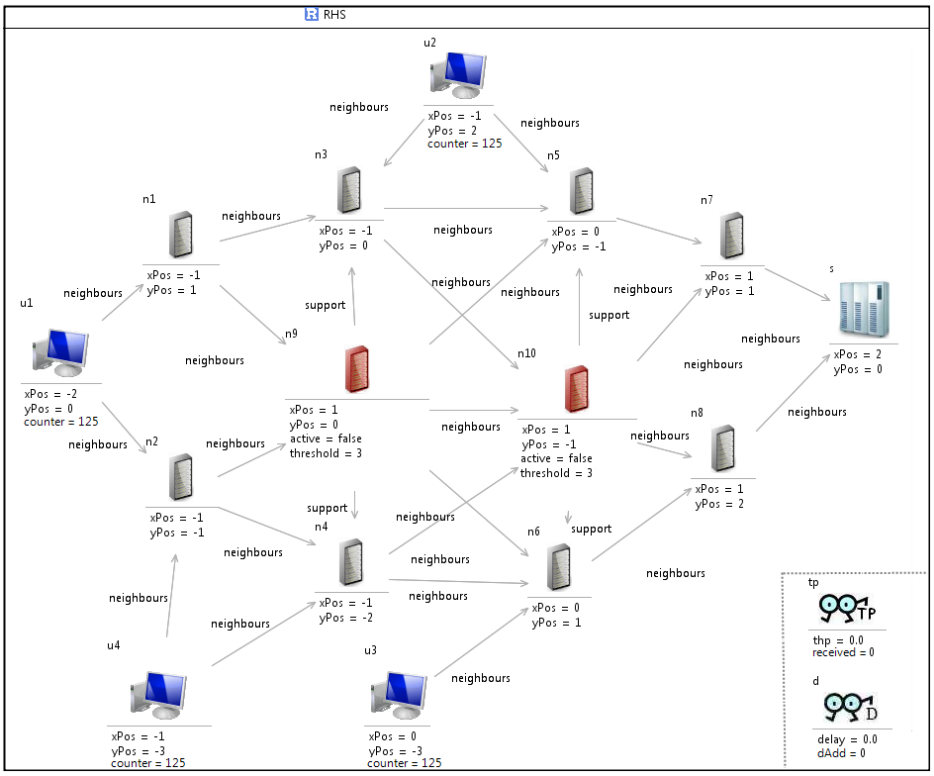


Figure 3.29: Initial model of the network.

- **Throughput** and **delay** of the overall network. They indicate how fast nodes process packets. Throughput tells us how many packets are processed by the network per unit of time. Delay indicates how many time units the packets spend within the network. The higher the throughput, the lower the delay, and so the higher the performance of the network.

- **Packets processed per node**. This measure provides an indication of the workload supported by each node. This is however a complex indicator due to the way in which packets are processed in this network, and how they arrive to nodes. The fact that processing time depends on the length of the buffer of pending packets may cause different behavior depending on whether packets are coming in bursts or at a regular pace.

- **Packets processed per SupportNode**. This measure is important because it provides an indication on the real need of these nodes.

- **Activation times of SupportNodes**. The time and frequency of activation of this kind of nodes also provides useful information about their actual usage in the current network configuration.

3.6.4 Step 5

The next step is to define observers according to the properties we want to monitor for our system. For that, we define the metamodel for the observers shown in Figure 3.30.

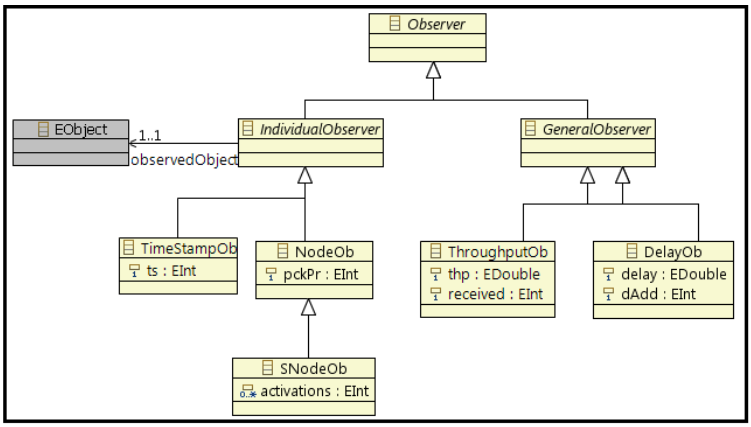


Figure 3.30: Observers Metamodel.

There are two different general observers:

- ThroughputOb, in charge of monitoring the throughput of the system (the number of Packets processed by the network per time unit).
- DelayOb, that tracks with its delay attribute the average time spent by packets to be processed by the network.

We also have three individual observers:

- TimeStampOb. An observer of this type will be associated with each Packet, and it will store the time the Packet arrives to the network.
- NodeOb. They will be associated to Nodes and will count the number of Packets processed by these.
- SNodeOb. This is an specialization of the previous one. An observer of this type will be associated with each SupportNode. It will count the number of Packets processed by them as well as the simulation times they activate/deactivate.

3.6.5 Step 6

Once the observers have been defined, we need to add the observers' behavior to the system behavioral rules defined in step 2. Since in step 2 we only defined one

system behavioral rule, we are defining here the remaining rules, already merged with observers.

Starting with the initial configuration of the system (Fig. 3.29), we see within the dotted area that we have included a general observer of each type in the network and initial values have been given to their attributes. Similarly, an observer of type NodeOb and SNodeOb is associated to each object of type Node and SupportNode, respectively, although they are not shown in the figure for space limitations.

The NewPacket rule, shown in Fig. 3.31, simulates the generation of Packets by a User as long as its counter attribute is higher than 0. This process follows a uniform distribution in the interval $[1,7]$, i.e., a User generates a Packet every duration time units. Here, duration determines the duration of the rule and is calculated using the random number generator available in *e-Motions*. The Packet's attribute is initialized at creation as shown in the rule's RHS. A new TimeStampOb observer is created and associated to the Packet.

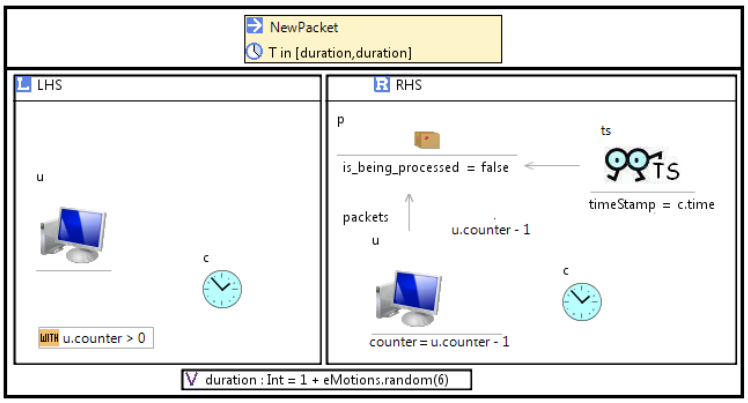
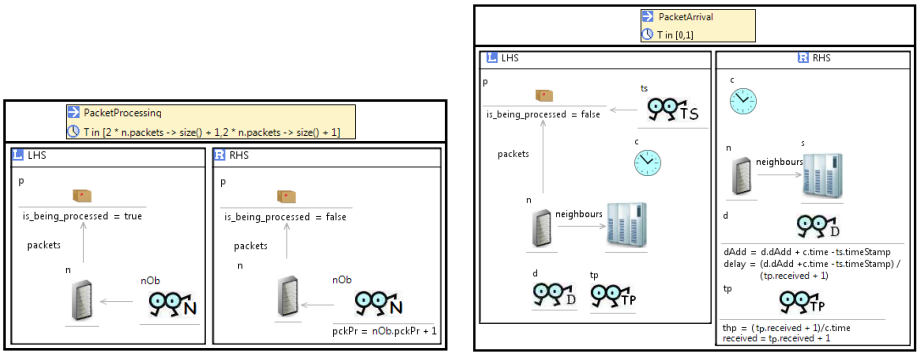


Figure 3.31: NewPacket Rule.

In Fig. 3.32(a) we can see the PacketProcessing rule. It models the processing of a Packet by a Node by modifying its is_being_processed attribute. The pckPr attribute of the observer associated to the Node is increased in one unit to indicate that the latter has processed a new Packet. The time this rule spends is directly proportional to the number of packets in the Node's buffer: the higher the number of Packets waiting to be processed, the more overloaded the Node gets and the longer the time to process one. PacketArrival rule (Fig. 3.32(b)) models the arrival and consumption of a Packet from a Node to the Server. The time this rule consumes is either 0 or 1 time units. In this rule, the two general observers are updated appropriately: the DelayOb observer updates its attributes to properly compute the delay, and the ThroughputOb computes the current throughput and updates the number of Packets received.

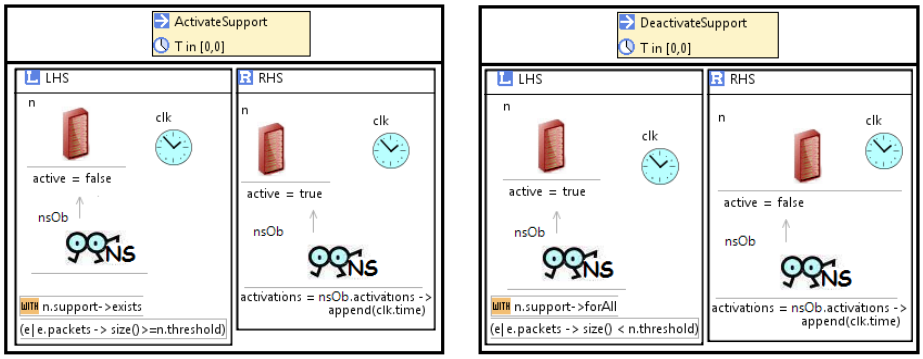
Finally, activation and deactivation of SupportNodes is specified by two rules. ActivationSupport rule (Fig. 3.33(a)) deals with the activation of a Sup-



(a) PacketProcessing Rule (b) PacketArrival Rule

Figure 3.32: PacketProcessing and PacketArrival Rules.

portNode when it is inactive and one of the Nodes it supports is processing more Packets than indicated by the SupportNode's threshold. DeactivationSupport rule (Fig. 3.33(b)) carries out the opposite action. In both rules, the time unit when the activation/deactivation occurs is added to the SupportNode's observer activations attribute. These rules are instantaneous rules, i.e., atomic rules with duration 0.



(a) Activate Rule (b) Deactivate Rule

Figure 3.33: Activate and Deactivate Rules.

3.6.6 Obtaining performance metrics

In order to understand how the network works and to analyse its performance parameters, we have simulated the network with different threshold values for the support nodes. They have ranged from -1 (the support nodes are always activate) to 100 (they will never be activate because the buffers of the nodes in our

example keep always below that value). For every threshold value we have run five different simulations since users introduce packets in the network in a random manner. The figures showed in the charts correspond to the average results for the obtained values. In all the simulations, we have limited the number of packets that enter the network and reach the server to 500 (each user introduces 125).

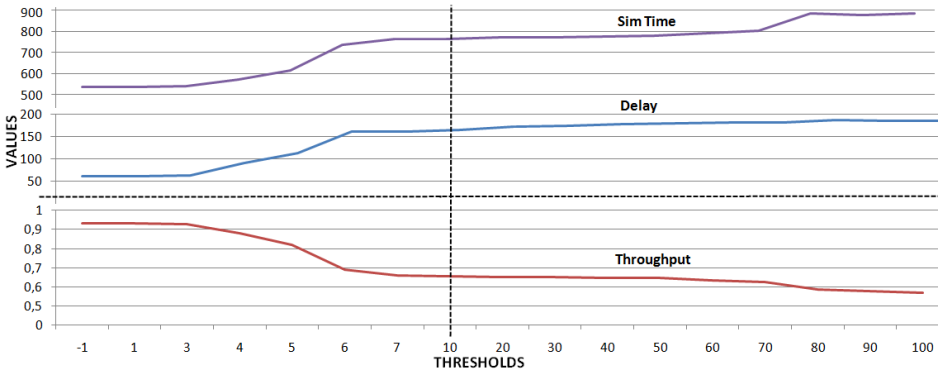


Figure 3.34: Simulation time, delay and throughput.

Fig. 3.34 shows the values of throughput, delay and the time units taken by the simulations. Most variations occur when the threshold is between -1 and 7, before they become stable. This is why the chart is divided in two horizontal parts, in which the left part zooms in the $[1, 10]$ interval. The vertical axis has also been split into two sections, in order to distinguish the area where the throughput values reside.

The examination of the chart reveals that, as expected, the best performance (highest throughput, lowest delay and lowest simulation time) is achieved when the support nodes are always active (threshold = -1). However, this is also the most expensive situation. The behavior of the support nodes turns out to be more interesting when the thresholds are between 3 and 6. In that range, the three parameters experiment the biggest variation, making the network slower as the thresholds increase.

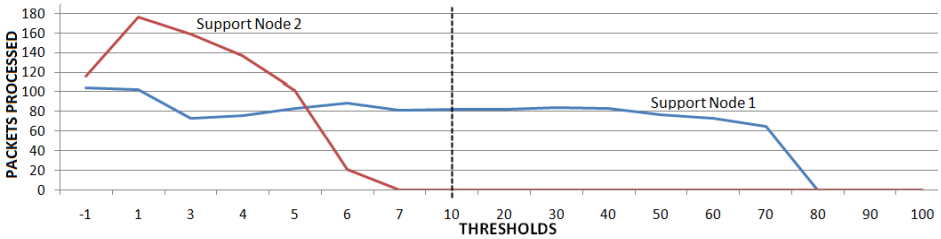


Figure 3.35: Packets processed by the support nodes 1 (n9) and 2 (n10).

Fig. 3.35 shows the chart with the number of packets processed by the two support nodes for each threshold value. We see that the second support node (node

n10 in Fig. 3.29) processes packets when its threshold is within the range $[-1,7)$. In fact, in the range $[-1,5]$ it processes more packets than the first support node (node n9 in Fig. 3.29). However, this latter node keeps on processing packets until the threshold is 80. These results were initially unexpected, and they are a result of the topology of the network and the way in which the packets are processed. By defining a different topology to the network and/or by changing the algorithm used to process packets to speed it up (i.e., changing the duration of rule PacketProcessing), the results would be different.

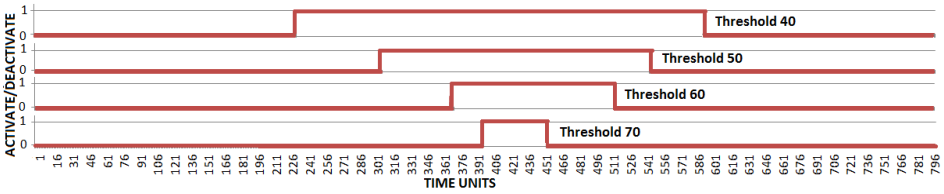


Figure 3.36: Activation/Deactivation of support node 1 (n9).

Focusing on the first support node (n9), it finally stops processing packets when the threshold value is 80. By looking at its behavior in the range [-1, 7], we cannot expect when this node would stop processing packets. In fact, looking at the behavior of the second support node (at the beginning it processes more packets than the first support node but then it stops processing packets from the threshold value 7), we may expect that the first support node will stop processing packets earlier than it actually does. To help us see how the activation/deactivation of a support node evolves, we have also displayed charts for it. Thus, Fig. 3.36 shows four charts with the activation/deactivation of the first support node when the threshold values are 40, 50, 60 and 70. We see how the activation of the node is carried out later when the threshold increases. This is because, as thresholds increase, the nodes being supported do not need the help of the support nodes so soon. Support nodes are also deactivated earlier when the threshold is smaller. We have to clarify here that the fact that a support node is deactivated does not mean that it stops processing packets. In fact, it only means that new packets stop coming in, but the node still has to process its buffer of pending packets.

Not all the graphs of activations/deactivations are as uniform as the four shown here [1]. For example, in Figure 3.37 we can see the times the support node n10 changes its state when thresholds are set to 3.

Finally, let us show a graph that we also consider of interest (Fig. 3.38). It displays the number of packets processed by two nodes supported by a support node (nodes n4 and n6 in Fig. 3.29). This graph is related to the one shown in Fig. 3.35, since the processing of packets by the support nodes makes nodes n4

[1] The complete set of charts and values obtained for the simulations can be consulted in the *e-Motions* web page devoted to this example: `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/PacketSwitchingExample/Results`

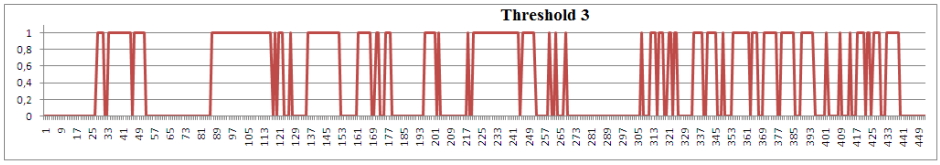


Figure 3.37: Activation/Deactivation of support node 2 (n10).

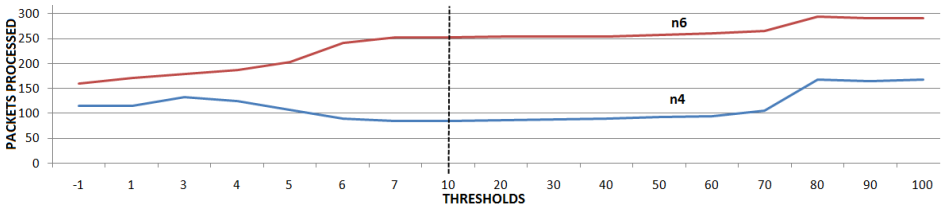


Figure 3.38: Packets processed by nodes n4 and n6.

and n6 process less packets. In general, we can see that the more packets the first support node (n9) processes, the less packets n4 processes, and the same thing happens with the second support node and n6. For every threshold value, n6 processes more packets than n4 because packets arrive to it from three ways, while they arrive to n4 from two ways.

Further examples developed in *e-Motions* that illustrate our proposal for modeling QoS properties of DSVLs can be found in [Ate11].

3.7 Summary

In this chapter we have proposed the use of special objects (observers) that can be added to the graphical specification of a system for describing and monitoring some of its non-functional properties, and shown its application to performance analysis of systems specified by user-defined DSVLs—in particular those whose behavior is specified in terms of in-place rules. Observers allow extending the global state of the system with the variables that the designer wants to analyze when running the simulations, being able to capture the performance properties of interest. Furthermore, the fact that action executions are first-class citizens in the *e-Motions* specifications [RDV09] can enable observers to monitor not only the states of the objects of the system but also their actions.

We have presented our approach through a case study of a hammers production line, a dynamic system where hammer's parts (and hammers themselves) flow among machines and containers. We have shown how, changing certain parameters of the system, the performance values obtained after simulation vary. Thus, the idea is to reconfigure the system until it achieves the expected quality of service (in this chapter we have focused on performance).

We have also extended *e-Motions* time modeling by allowing the use of several

probabilistic distributions. This permits us model many kinds of systems.

We have described a general methodology for the performance analysis of systems using our approach, presenting a series of steps and making emphasis on how simulations can be conducted and the kinds of analysis that are possible with our proposal. We have also shown the tool support for our approach and discussed the pros and cons of it. Finally, we have applied this general methodology to develop the packet switching case study. Further examples and case studies can be found in [Ate11].

4

A DSVL for Modeling Power-Aware Reliability in WSNs

As it has already been explained in this dissertation, the modeling and analysis of non-functional properties is very important when developing different kinds of systems. Software quality assessment is often applied at system implementation time, which is normally too late because the problems arisen during implementation can force the re-engineering of important parts of the system, which is very costly. This is why it should be raised to the system design phase. In this chapter we focus on a non-functional property of systems known as *reliability*, and we try to measure it at design time. Concretely, we analyze reliability when modeling systems based on components by means of domain-specific visual languages (DSVLs), key parts in Model-Driven Engineering (MDE) for representing models and metamodels. In this way, we pursue the correct and complete specification of a system by including the specification and analysis of its reliability properties at design time. Specifically, here we focus on system reliability in the context of wireless sensor networks (WSNs) and energy consumption.

In order to obtain reliability measures for a system, it is necessary to be clear about what reliability is and how it is defined. Different definitions have been given for it. For example, the *Wikipedia* defines it as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time". In [FP98], Fenton and Pfleeger say that "the accepted view of reliability is the probability of successful operation during a given period of time". Musa provides in [Mus98] a similar definition: "reliability for software products is the probability for the software to execute without failure for some specified time interval". A more general definition of reliability is given by the quality model in ISO 9126-1 [ISO01]. There, reliability is defined as "the capability of the software product to maintain a specified level of performance when used under specified conditions". One more definition, in this case for software reliability, is given by Wohlin et al. [WHRW01]: "the probability for failure-free operation of a program for a specified time under a specified set of operating conditions".

A WSN is made up of spatially distributed sensor nodes deployed over a certain area to monitor physical or environmental conditions, such as sound, pressure, temperature, vibration, humidity, and to cooperatively pass their data among the

nodes. The range of applications of WSNs is large, and it includes military operations, habitat and environmental monitoring, area surveillance or remote sensing. Reliability is a crucial aspect in WSN applications, especially those deployed for real-time communication, since data delivery should be guaranteed. For this reason, it is very important the routing protocol chosen in each circumstance. There are many routing protocols already studied [GT12], which can be classified in broad terms as fault-tolerant routing [Dat03], geographic routing [FS05] and energy aware routing [SR02, TEO05]. In this chapter we focus in the last group, and concretely in the Directional Source Aware routing Protocol (DSAP) [JN10] in order to study its reliability, at design time, in the original implementation and some variants.

The DSAP and its variants aim to extend the life of the network as much as possible. In this sense, there is a trade-off between the network's lifetime and the time taken for packets to reach their destination. Regarding reliability and the definitions given above, we are going to model and monitor reliability in terms of the network's lifetime. Thus, a protocol will be more reliable than another if it keeps the network working longer. This is in accordance with the definitions of reliability given by Fenton and Pfleeger [FP98] and Musa [Mus98]. Throughout this chapter, we will be presenting the DSAP and some variations for its routing. We model the DSAP and its reliability in the domain of MDE and in-place rules based on DSVLs. For the implementation and simulation, we use *e-Motions*, the graphical framework and tool for defining timed behavioral specifications of models explained in Section 2.2. We show as well how different kinds of reliability analysis can be carried out and perform experiments based on realistic situations.

In this chapter, we also take a step forward in the high-level modeling of systems compared with the approach presented in the previous chapter. In the production line case study, hammer's parts (heads, handles and hammers) are modeled as class in the metamodel, so they appear as objects in the models and behavioral rules. Same thing for packets in the packet switching case study. According to this, the more hammer's parts and packets we have in these systems, the bigger are the models (because they are composed of more objects), so the slower the simulations are. This is because in Maude, the bigger a model is, the more expensive it is to realize a matching in a conditional rewriting rule. For this reason, in this chapter we model the objects that flow within the system, packets in the case of WSNs, as class attributes instead of classes themselves. We shall see this approach in Section 4.2.2.

After this introduction, in Section 4.1 we present an approach for modeling reliability of systems based on components in design phases. In Section 4.2 we present a DSVL to model the DSAP, some variants of it, and to monitor and analyze its reliability. Section 4.3 summarizes the chapter.

4.1 Modeling Reliability in DSVLs

4.1.1 Modeling Behavior

As presented in Section 2.2, we specify the dynamic behavior of a DSVL by describing the evolution of the modeled artifacts along some time model. We achieve this by applying model transformations supporting in-place update. In this way, a system starts executing in a particular state, and it evolves over time by the non-deterministic firing of the behavioral rules.

4.1.2 Probabilistic Rules

In many kinds of systems, state machines are good for defining the state of their components. If these state machines only have two states, the transition from one state always brings to the other one. However, in the case of components having more than one possible state, their state machines can transit from one state to more than one other state. Our in-place rules are triggered when there is a match of their LHS in the system. We want to model that the triggering of a rule can make the system evolve to more than one possible state, as it happens in state machines, according to a given probability for each transition. This is specially useful when we are interested in modeling the reliability of systems.

Let us first introduce the concepts of *mean time to failure* (MTTF) and *mean time to repair* (MTTR). The former represents the average from the time that a component of the system is put into service until it first experiences failure, while the latter is the average time that the component is out of service before it is repaired. As explained in [Pag89], both times are normally modeled with exponential distributions. We presented in Section 3.4 how we are able to apply many probabilistic distributions to our in-place rules' internal variables and durations. In this way, to model the MTTF of every component in our system, we only need a very simple rule where there is a component in its LHS which changes its state in the RHS. The duration of the rule will follow an exponential distribution whose parameter is the mean time to failure of the component. Figure 4.1 shows this rule in a system where the components are chips. A similar rule would be needed for modeling the MTTR.

Since we want to find a way of modeling systems whose components can have more than two states, we introduce in-place rules with more than one RHS. Imagine we have a system made up of chips, where each one can be in one of three states: fully working (fw), partially working (pw) or down (d). When a chip is in any of the three states, it can transit to one of the other two with a given probability (see Figure 4.2 for an example).

Now, to model the failure of a chip, we have to consider that it can evolve from the fully working state to either partially working or down states. The rule modeling this has two RHSs, and it evolves to one of them according to a probability. As before, the duration of the rule is exponentially distributed. Consequently, the

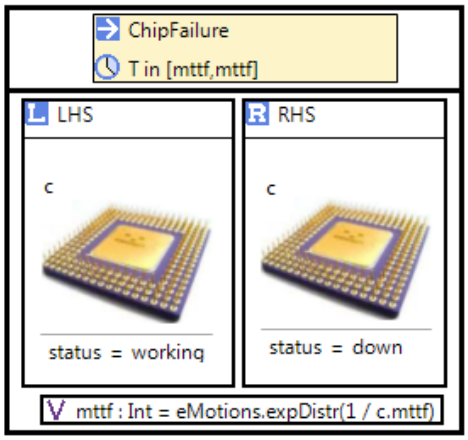


Figure 4.1: ChipFailure Rule

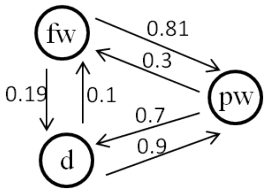


Figure 4.2: Transitions and probabilities

rule is "twice" probabilistic: (1) due to the probability of transitions and (2) to the exponential duration. Such rule is shown in Figure 4.3. It is very simple, and since the only difference among its two RHSs is the value of the status attribute of the chip, it could be modeled with only a RHS and an OCL *if* condition in that attribute. However, this same approach can be used to create rules whose several RHSs differ completely.

4.2 Reliability in WSNs

In this section we apply our approach for specifying and measuring the reliability of systems modeled with DSVLs. Concretely, we model the structure and behavior of a routing protocol for WSNs named DSAP.

4.2.1 The DSAP

The Directional Source Aware routing Protocol (DSAP) [SWKS01] was designed for low-power fixed wireless topologies and based on local information where each node only knows about its neighbors information. It has several advantages over other routing protocols, including incorporating power considerations and having no routing table [SWKS01]. Each node has a unique ID, which gives how

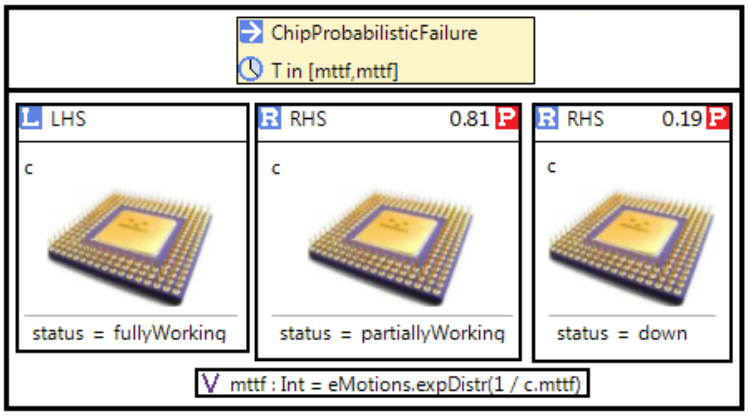


Figure 4.3: Rule with two RHSs

far, in terms of number of nodes, the node is from the network perimeter in each direction. For example, the ID of the node numbered 43 in the WSN in Figure 4.4 is $(3,3,4,4,6,5,5,3)$. This means that there are three nodes (42, 41, 40) to the edge in direction 0 (left), three nodes (32, 21, 10) in direction 1 (up-left), etc. Consequently, the ID is a vector with as many components as neighbors have the nodes in the network – 8 in our case.

When transmitting a packet, each node contains information about its neighbors' IDs and the ID of the packet's target node. In order to choose which neighbor a packet is forwarded to, the Directional Value (DV) is used. The DV of each neighbor is calculated by taking their IDs and subtracting them from the destination node's ID. Let us imagine that node 22 wants to send a packet to node 77. The original DSAP can be seen as a two-step routing protocol.

First, the source's and target's IDs are subtracted. The result in our case is $(2,2,2,2,7,7,7,2) - (7,7,7,2,2,2,2,2) = (-5,-5,-5,0,5,5,5,0)$. This obtained vector indicates which neighbors the packet can be forwarded to: those with a non-positive number are discarded. In this way, the packet will not be sent to nodes in directions 0 (left – 21), 1 (up-left – 11), 2 (up – 12) and 3 (up-right – 13). Second, the ID of those candidate neighbors to receive the packet is subtracted from the destination node's ID. The absolute values of the components in each resulting vector are added, which gives us the DV, and the neighbor with the smallest result is chosen. If there are more than one with the same value, one of them is randomly selected. In our case, the DVs for the nodes numbered 23, 33, 32 and 31 are, respectively, 28, 26, 28 and 32. As it was obvious, node 33 is closer to the destination and is the one chosen.

Variants. There have been some new routing methods proposed for improving the DSAP in order to extend the sensor network's lifetime, such as the power aware routing [SS04]. However, in every proposed routing, the neighbor node which has the most power and shortest path is chosen most of the time. This causes

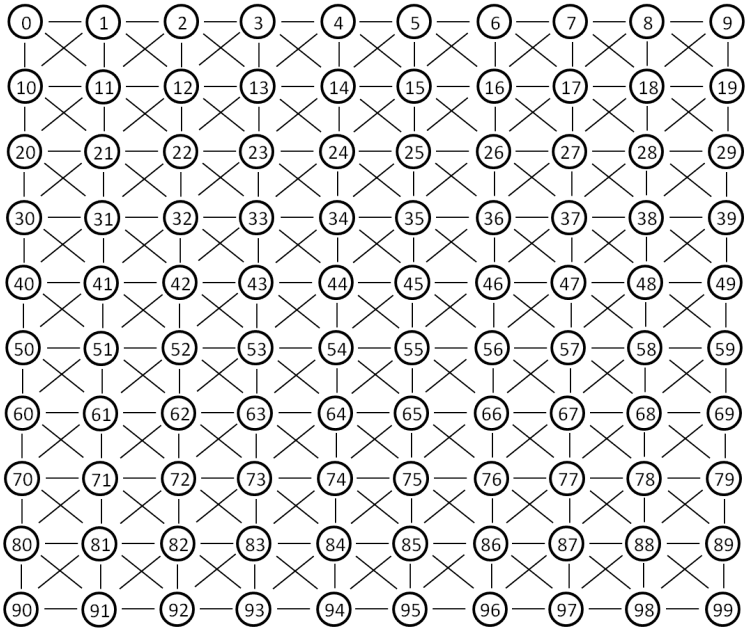


Figure 4.4: 100 nodes WSN

the energy in the same nodes to be depleted, and creates an unbalanced power dissipation in the network. Besides, since the protocol always tries to forward the packet to a neighbor closer to the destination, some of the nodes in the network will stay untouched, whereas they could be chosen as an alternative path to prolong the overall network lifetime.

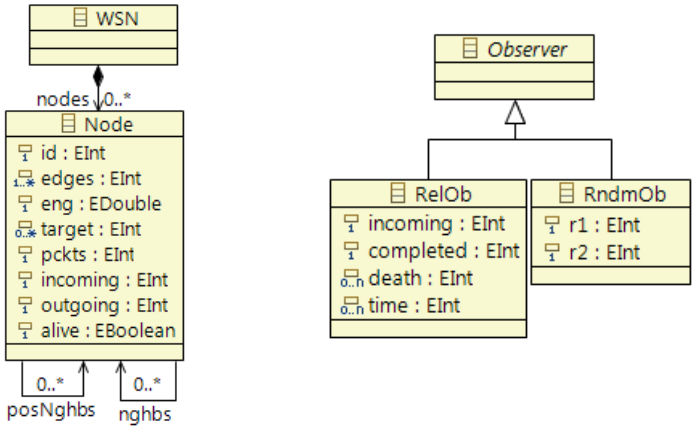
Here we select some protocols already proposed and consider a variant of the power-aware DSAP. Furthermore, we consider all possible directions from a given node, even if in the original DSAP the substraction of the source node's ID and target node's ID gives a negative number in a given direction. In this way we try to consider all possible paths and extend the network's lifetime as much as possible.

4.2.2 Modeling the DSAP and its Variants

This section presents how we model the DSAP, its variants and their reliability in terms of energy consumption, and explain how they could be extended and made more real with the approach presented in Section 4.1. We follow an MDE approach and propose the specification of a DSVL for the high-level modeling and analysis of the protocols, in terms of behavioral rules, in the design phases. We show how once we model the original DSAP, modeling each of its variants is trivial.

Defining the Metamodels

The first step is to define a metamodel for the DSAP in WSNs, which describes their static structure. Our proposed metamodel is the simple one shown in Figure 4.5(a). According to it, a WSN is composed of a set of Nodes. Each Node has an identifier given by an integer (id – 0, 1, 2 and so on). Another identifier, which gives the position of the node in the network according to what was explained in Section 4.2.1, is kept in the edges attribute. It is a sequence with n components in a n-neighbors network. The remaining energy is kept in eng. The attribute named target contains the edges of the target(s) node(s) and pckts contains the number of packets that a node currently contains. These two attributes work like this: if there are no packets in a node, target is an empty sequence; if there is one packet, target contains a sequence with n components (in a n-neighbors network); if there are more than one packet, target contains a vector with $n * pckts$ components, where the first n components represent the edges of the target node of the packet that arrived first, and so on. This is how we achieve to model packets as attributes instead of classes, as mentioned in the introduction. This improves simulations performance. Attributes incoming and outgoing contain the total number of incoming and outgoing packets in the node, and alive is a boolean stating if the energy of the node is above 0 (alive=true) or not (alive=false). If a node is not alive, it can neither receive messages nor forward them. As for the references, every node has a link (nghbs) to each neighbor and another link only to the neighbors which are alive (posNghbs).



(a) WSN Metamodel. (b) Obs Metamodel.

Figure 4.5: Metamodels

With the metamodel described we can already define behavioral rules for the functional behavior of the DSAP. Nevertheless, let us first present the Observers metamodel in order to be able to introduce observers in the rules and monitor

reliability properties. It is presented in Figure 4.5(b). There are two observers, named RelOb and RndmOb. The former keeps the number of packets that arrive to the network (incoming) and those that reach their target node (completed). It also contains two sequences, death and time, that store the identifier of the nodes which die and the time they die. The latter observer is used to randomly select nodes from and to which packets are sent.

Defining the Behavior

Here we define the behavior of the DSAP for 8-neighbors WSNs as that presented in Figure 4.4. Modeling the behavior for a n-neighbors WSNs is trivial. We want to carry out two different analysis, one modeling the reality and another one for a more specific study. In the former, packets arrive at nodes randomly selected and their target node is also random. In the latter, packets are always sent from node 22 to node 77 (Figure 4.4).

Regarding the energy of the nodes and the size of the packets, we use a simple radio model described in [JN10]. In such model, the radio dissipates $E_{elec} = 50nJ/bit$ to run the transmitter or receiver circuitry and $E_{amp} = 100pJ/bit/m^2$ for the transmit amplifier to achieve an acceptable E_b/N_0 (see Figure 4.6 and Table 4.1) [HCB00]. To transmit a k-bit message a distance d meters using this radio model, the radio expends $E_{Tx}(k,d) = E_{Tx-elec}(k) + E_{Tx-amp}(k,d) = E_{elec} * k + E_{amp} * k * d^2$. To receive this message, the radio expends: $E_{Rx}(k) = E_{Rx-elec}(k) = E_{elec} * k$.

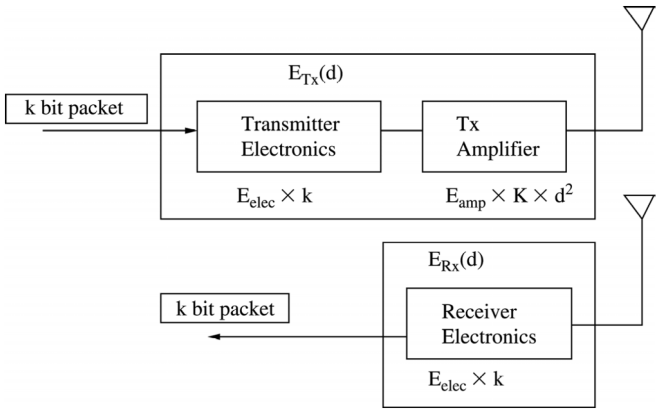


Figure 4.6: First-order radio model (from [SS04])

We assume as well that the distance between the wireless nodes is equal to each other and all data packets contain the same number of bits. The parameters are the following: distance $d = 0.5m$ and number of bits transmitted $k = 512bits$. In short, every node dissipates 25612.8 nJ in the transmission of a packet and 25600 nJ in its reception. We suppose that every packet starts with an energy of 1000000 nJ.

Table 4.1: Radio Characteristic (from [HCB00])

Operation	Energy Dissipated
Transmitter Electronics ($E_{Tx-elec}$) Receiver Electronics ($E_{Rx-elec}$) ($E_{Tx-elec} = E_{Rx-elec} = E_{elec}$)	$50nJ/bit$
Transmit Amplifier (E_{amp})	$100pJ/bit/m^2$

Our initial model is the network composed of 100 nodes shown in Figure 4.4. There are also a RelOb and a RndmOb observers. We have a rule named CalculateRandom where it is decided the source and target nodes for the next incoming packet. Their ids are kept in the observer's attributes r1 and r2. Rule PacketArrival (Figure 4.7) uses the values in such attributes to model the arrival of the packet. It arrives to node n0, while its target is node n1 (LHS). Notice the corresponding update of the necessary attributes in the rule's RHS.

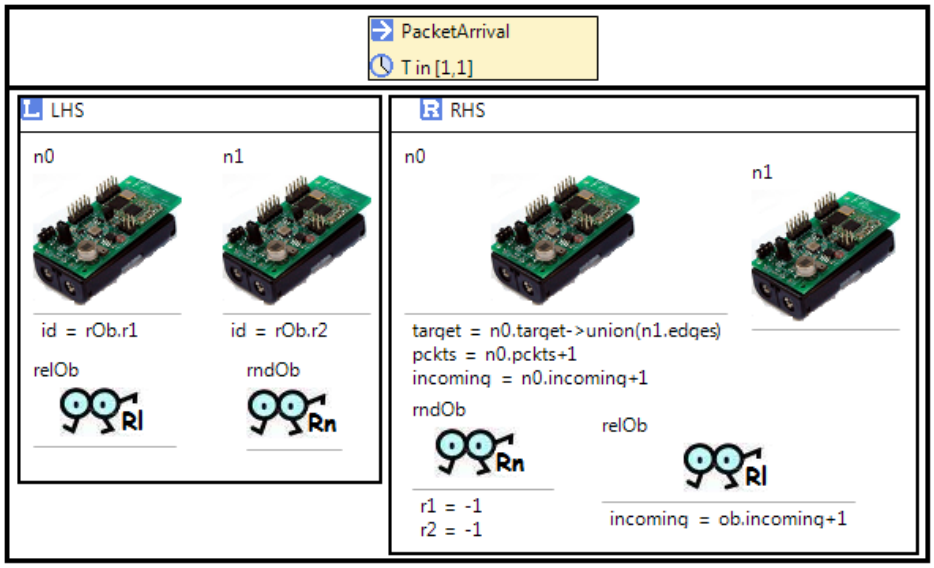


Figure 4.7: PacketArrival Rule

Rule PacketForwarding (Figure 4.8) models the forwarding of a packet to the alive neighbor with the lowest DV. The OCL condition in the LHS checks that node n0 has a packet and n1 is a neighbor with positive energy and the lowest DV. That expression uses a helper, named dv, which deals with the calculation of the DV and is: context Sequence::dv(s1 : Sequence, s2 : Sequence): Integer body: self -> iterate(i ; acc : Sequence = Sequence{} | acc->append(i.abs()))->sum(). In the rule's RHS, the attributes of the nodes are modified with the updating of the energy, the number of packets contained, incoming and outgoing packets, etc.

That way we model that an actual packet leaves node n and gets into node textsfn1. The RelOb observer updates its completed attribute when n1 is the target node of the packet being forwarded.

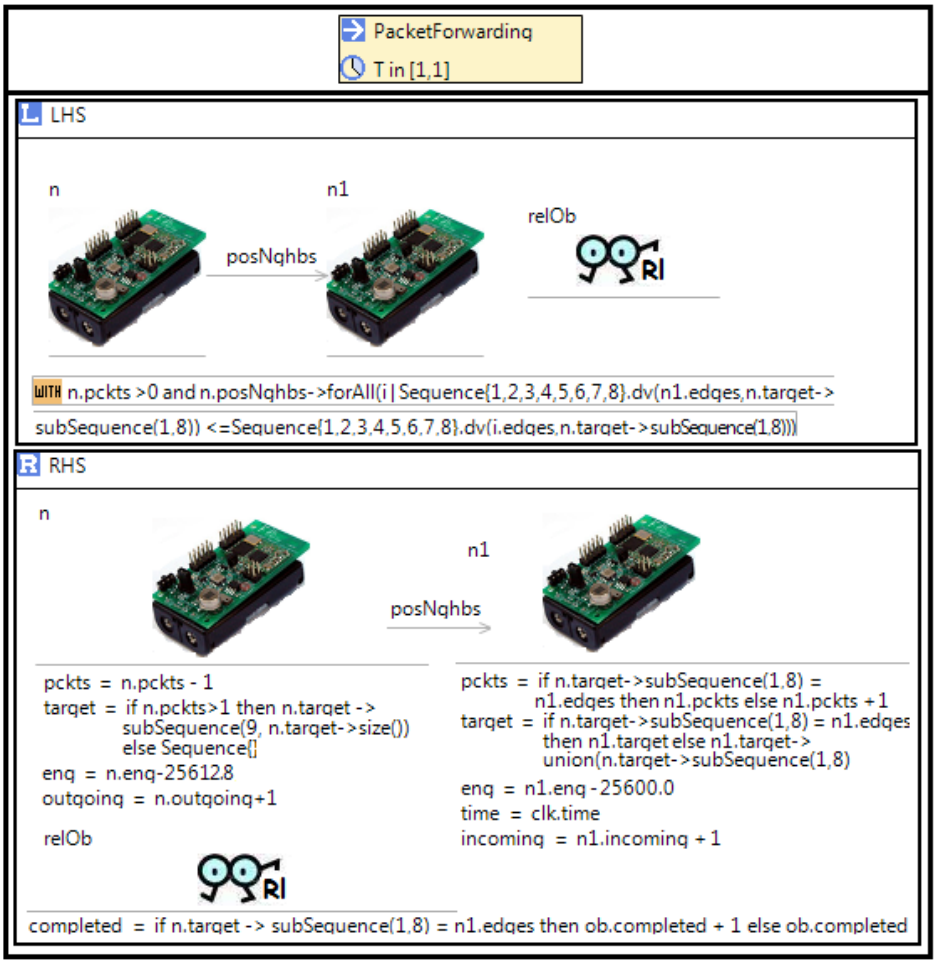


Figure 4.8: PacketForwarding Rule

To model some of the variants of the DSAP, we only need to change the OCL condition in the PacketForwarding rule's LHS. For example, we can model the Power-DSAP with power-aware routing [SWKS01]. It selects the paths according to the ratio of the directional value and the power available at the neighboring nodes. The OCL condition would be this: n.pckts > 0 and n.posNghbs -> forAll(i | Sequence{1,2,3,4,5,6,7,8}.dv(n1. edges, n.target -> subSequence (1,8)) / n1.eng <= Sequence{ 1,2,3,4,5,6,7,8}.dv(i.edges, n.target -> subSequence(1,8)) / i.eng). We have developed another variant, where the routing selects the path according to the ratio of the DV, the power remaining at the neighboring nodes and it also takes into account the packets contained at the

neighboring nodes and the power they will spend in forwarding them. The OCL condition of the PacketForwarding rule's LHS would be the following: n.pckts > 0 and n.posNghbs -> forAll(i | Sequence{1,2,3,4,5,6,7,8}.dv(n1.edges, n.target -> subSequence (1,8)) / (n1.eng - n1.pckts* 25612.8) <= Sequence{1,2,3,4,5, 6,7,8}.dv(i.edges, n.target -> subSequence(1,8)) / (i.eng - i.pckts*25612.8)).

Regarding the rules' duration, notice that we have established that a new packet arrives every time unit and nodes forward packets (as long as they have any) every time unit too. To make the model more realistic, these times should have followed some probabilistic distribution, something that we are able to model as we already presented in Section 3.4. However, to make the simulations the least random possible, we decided to set these times as 1 in order to compare the different protocols in the different simulations in a fair way.

The complete specification of the DSAP can be consulted at [Ate12]. It contains for example a rule to keep in the RelOb a list with the death nodes and the time they die.

Extensions to the DSAP. In order to make the modeling and simulations more realistic and according to real-life situations, we can add a few rules for that purpose. In real-life WSNs, nodes consume energy when they are in stand-by. The failure of nodes can also be modeled, with rules similar to those in Figures 4.1 and 4.3. For such purpose, we have run experiments where we model power consumption of nodes in stand-by and also the death of nodes due to other circumstances. They are explained in the next section. Likewise, we can model the repair of nodes which run out of energy or simply break.

4.2.3 Reliability Analysis

Once we have the specifications of the protocols modeled in *e-Motions*, we are able to simulate them and analyze their results. These specifications are automatically transformed to a domain with well-defined semantics, namely Real-Time Maude [ÖM07], by means of ATL [JABK08b] transformations. We show results with a 100-nodes and 8-neighbors network, such as the one evaluated in [SS04], and with a 12-nodes and 4-neighbors network, like the case study used in [JN10].

100-nodes and 8-neighbors network

As we mentioned in Section 4.2.2, we want to carry out two different analysis. In one of them, the source and target nodes of packets are chosen randomly, while in the other packets always go from node 22 to node 77 (Figure 4.4). The first one models more accurately the reality, while the second one allows us to focus on nodes 22 and 77 and their neighbours. For both kinds of analyses, we have run three protocols: (1) the original DSAP [JN10], (2) the power-aware DSAP [SS04] and (3) a new variant that also takes into account the packets that still need to be processed in each node apart from the remaining energy. Let us call the last

protocol power-aware DSAP v2. Recall that in the three protocols we consider all the neighboring nodes as candidate nodes to forward a packet and not only those whose subtraction with the target node is positive.

Random Arrivals

In these simulations, the RndmOb observer deals with the random selection of the source and target nodes for every packet. We are interested in knowing the lifetime of the network in each protocol. For that, we consider that the network dies when a node consumes all its energy. So we will measure the reliability in terms of the time the network has been alive and the number of packets completed (those that reached their destination) within that time.

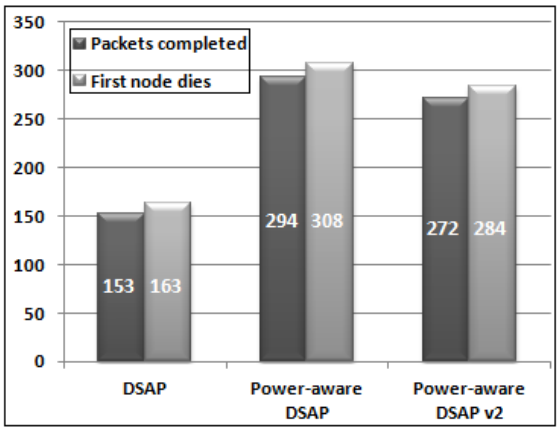


Figure 4.9: Results with random arrivals

As we already knew from other works [SWKS01, SS04], and as we can see in the simulation results shown in Figure 4.9, the power-aware DSAP is more reliable than the original DSAP. Besides, it turns out that it is also more reliable than the new developed protocol that also takes into account the packets that still need to be processed in the nodes.

Fixed Arrivals

We have made these simulations stop when all the energy in the nodes around 22 or 77 is consumed, so that no packet can reach its destination (77). We have made node 22 not to consume any energy when forwarding packets and node 77 when it receives packets. Otherwise, the energy in these nodes would be consumed very soon.

In Figure 4.10 we can see the time at which the energy in each node around node 22 is consumed for each protocol. Let us focus first in the original DSAP. Since it does not take into account the remaining energy in the nodes, it always follows the shortest path to the destination. This is why it always uses node 33 from 22 to send packets to node 77 at the beginning. This makes the energy in node 33 to be consumed quickly, in time 22. Then, the protocol forwards packets to nodes 23 and 32 from 22 because they are at the same distance from

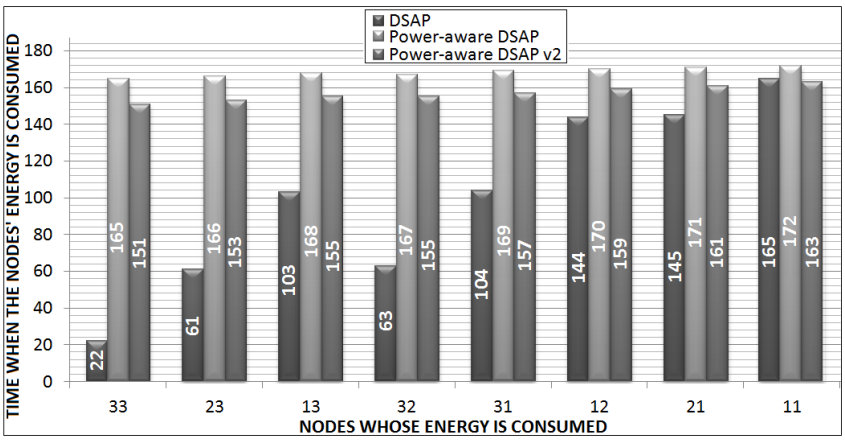


Figure 4.10: Results with fixed arrivals

the destination. The energy at both nodes is consumed almost at the same time, at times 61 and 63, respectively. The same happens then with nodes 13 and 31, and later with nodes 12 and 21. The last node to run out of energy is the 11. The number of packets that reach node 77 are 87.

In the other two protocols, the energy consumption in the nodes is much more uniform, since they take into account the remaining energy at nodes. In the power-aware DSAP, the energy consumption is slightly more uniform than in the power-aware DSAP v2, and the packets completed are 128, while in the latter protocol they are 123.

From the three analyzed protocols, the power-aware DSAP already presented in [SWKS01, SS04] is the most reliable both with random and fixed arrivals.

12-nodes and 4-neighbors network

Now we make experiments with the 12-nodes and 4-neighbors network shown in Figure 4.11. These experiments are more realistic since we consider power consumption of nodes in stand-by and failure of nodes due to other circumstances apart from energy depletion. In fact, WSNs can be deployed in many and different environments, so there are different causes that can make nodes die. For example, animals could try to eat them, or they could be damaged due to adversary environmental conditions.

The chart in Figure 4.14 displays three different executions in such network:

- The blue one (*Power-DSAP*) is an execution with the Power-DSAP [JN10]. Recall that every node dissipates 25612.8 *nJ* in the transmission of a packet and 25600 *nJ* in its reception.

- The red one (*Power-DSAP SB consumption*) uses the Power-DSAP and it also considers the power consumption of nodes in stand-by. Concretely,

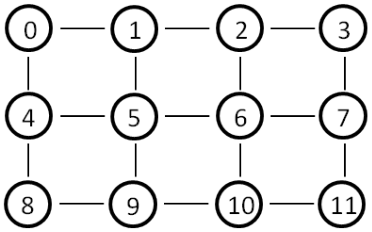


Figure 4.11: 12 nodes WSN

every node consumes 3000 *nJ* every unit of time. The rule used to model this is the simple one shown in Fig. 4.12.

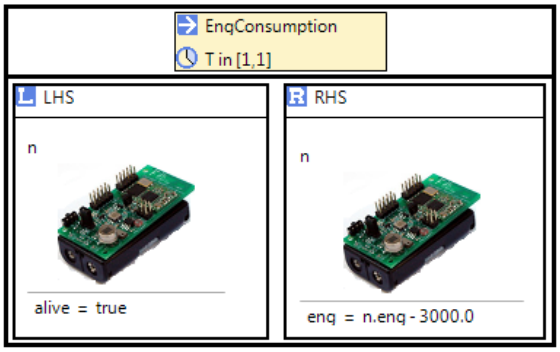


Figure 4.12: Power consumption in stand-by

- The green one (*Power-DSAP exp death*) applies the Power-DSAP and it also considers the failure in nodes due to other circumstances apart from the energy depletion. Such failure follows an exponential distribution of mean 200 units of time. This concept of "random" failure or death is the same as the *mean time to failure* presented in Section 4.1.2. To model such death, the rule shown in Fig. 4.13 has been used, which follows the pattern presented in Fig. 4.1. Note that the RelOb observer has a new attribute, eng, apart from those shown in Fig. 4.5(b). It is of type Sequence and we use it to store the remaining energy in nodes when they die.

The X axis in the chart shows the simulation time, while the Y axis indicates the number of nodes alive. It can be seen as well the concrete nodes that die (indicated by the numbers on the lines). In the case of the *Power-DSAP exp death*, it is indicated between brackets the energy available in the nodes when they die. In the three cases, packets arrive at random nodes, and their destination is also random. Let us analyze each of the three cases.

In the Power-DSAP, the first node (number 4) dies due to energy depletion in time 82. From that moment, several nodes die, until node number 6 does in time

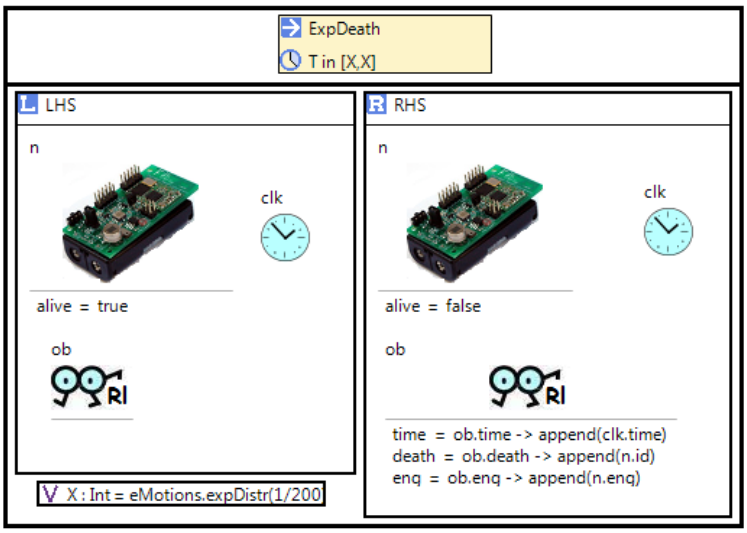


Figure 4.13: Death due to random circumstances

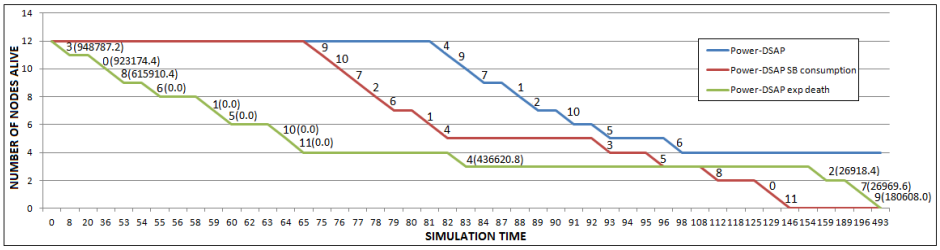


Figure 4.14: Death of nodes along time

98. From that moment, the nodes alive are 0, 3, 8 and 11, which cannot exchange packets among them, so there is no packet flow any longer. That is the reason why no more nodes die from time 98. The number of packets that arrive to their destination is 83.

In the Power-DSAP with power consumption of nodes in stand-by, the first node (number 9) dies due to energy depletion in time 71. It happens earlier in simulation time than in the previous case due to energy consumption in stand-by. From that moment, several nodes (10, 7, 2, 6, 1 and 4) run out of energy until time 82. At that point, only nodes 0, 3, 5, 8 and 11 are alive. They cannot exchange packets between them because they are not directly connected. This is why they die later in time: the remaining energy in those nodes from unit of time 82 is depleted only due to power consumption in stand-by, being no packet flow any longer. The last node to die is number 11 in time 146. 75 packets arrive to their destination in this case.

In the Power-DSAP that considers the death of nodes due to other circum-

stances apart from energy depletion, the first node (number 3) dies in time 8. When it dies, its energy is 948787.2 nJ, so it dies due to a random circumstance. The same happens with nodes 0 and 8, which also die quite early in time. Having three less nodes already in time 53, the remaining nodes have to work more. This causes the energy in node 6 to be depleted in time 55, and the energy in nodes 1, 5, 10 and 11 to be depleted in times 59, 60, 64 and 65, respectively. From unit of time 65, the only nodes alive are 2, 4, 7 and 9. Their energy is not consumed anymore since they cannot exchange packets, so all of them die due to other circumstances. The last one to do it, is node 9 in time 493. The number of packets that arrive to their destination is 51.

With these experiments we have shown that we are able to model and simulate real-life scenarios that were not considered in the papers that dealt with the DSAP [JN10, SWKS01, SS04]. The values chosen for quantity of energy consumption per time unit and the distribution followed for the failure of nodes due to random circumstances could have been different depending on the network. In any case, our experiments serve as proof of concept for our approach.

4.3 Summary

In this chapter we have presented an MDE approach to perform a high-level modeling of the reliability in systems based on components by describing how the state of their components can vary. We have introduced the use of several RHSs in transformation rules for describing the behavior of components in terms of their reliability. We have also presented how we can easily model the DSAP for WSNs and some variants by simply realizing small changes in the rules. By using observers we have been able to monitor and analyze the reliability of the protocols in terms of energy consumption in nodes. Furthermore, we have described how we extend the modeling of the protocols by including some behavioral rules that allow more precise and realistic simulations by modeling energy consumption of nodes in stand-by or how nodes fail due to random circumstances.

5

Specification and Simulation of QNMs using DSLs

Expressiveness is the ability of a language to concisely and readily represent the desired domain concepts. Expressiveness is related to both the expressive power of a language, which describes the ideas expressible in that language (and those which are not), and how easily and precisely these ideas can be represented and manipulated. In computer science, each language (and associated notation, underlying logic and set of supporting tools) is more appropriate for expressing specific system aspects. For example, Java is good for describing systems in an object-oriented style and for executing them; Ecore and Meta-Object Facility (MOF) are good for representing the structure of metamodels; Query/View/Transformation (QVT) and ATL Transformation Language are well suited for transforming models; Petri Nets allow modeling and simulating concurrent systems; Queuing Networks were conceived for conducting performance analysis, etc. Each one is more appropriate to represent and reason about certain properties, and to conduct certain kinds of analysis.

Since no language fits all purposes, one widely adopted solution by software engineers is to build semantic bridges between modeling spaces [DGFD06], so that programs or models can travel from one to the other, being able to use the local analysis tools to reason about their properties, conduct simulations, validations, etc [Val08]. Bridges can also be very useful to provide semantics to those metamodels that lack precise semantics (e.g., UML). In these cases, the semantics of a set of source elements is given by the semantics of the corresponding elements in the target domain.

In Model-driven engineering (MDE), semantic bridges can be naturally specified by means of model transformations [Val08]. Thus, there are already many interesting proposals for bridging UML models to several semantic domains for performance and reliability analysis [BDIS04].

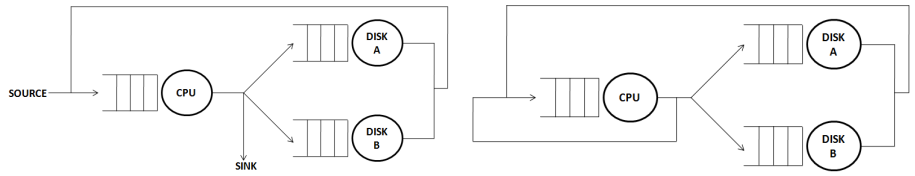
Bridging modeling domains is easier when they have similar expressiveness, because in this case model transformations can be naturally defined between them. However, the situation is not so clear when the expressiveness of the two domains to relate is rather disparate. In fact, the expressiveness of two languages can be compared by checking whether all valid sentences (or models) in one language

can be represented in the other, and by analysing how complex the relationship between these two representations is.

In our case, we were originally interested in bridging our notation and associated toolkit for real-time systems modeling, *e-Motions* [RDV09, Ate11], to other notations and tools. The goal was to be able to conduct different kinds of analysis on the *e-Motions* models, beyond the ones allowed by our tool. One of our choices was Queueing Networks Models (QNMs), given its possibilities for conducting performance analysis of the models in an analytical way [DB78]. However, we soon discovered that most of the information available in the *e-Motions* models was lost or severely demeaned when translated into QNMs. For example, the (rather complex) OCL expressions with the algorithms for selecting the next elements to move in some of our systems were reduced to just probabilities, which had to be estimated somehow (normally using simulations, which were precisely the analysis we tried to avoid by the use of analytical methods). Similarly, determining the *e-Motions* model elements that should be transformed into servers, jobs or queues was not a trivial task, and even not possible in some situations without human intervention.

We soon discovered, however, that the reverse mapping was not only feasible but even easy to implement. Building a bridge between QNMs and *e-Motions* would allow using the *e-Motions* simulation and analysis tools with QN models, bringing along several interesting benefits. For instance, we could easily support arbitrary probabilistic distributions for arrival and service times, beyond the usual Exponential and Uniform distributions; we could also leverage the advantages of MDE for building, in a high-level and modular way, a domain-specific language (DSL) for modeling QNMs and a supporting toolkit for simulating them; we could use (an Ecore version of) PMIF [Smi10], a performance model interchange format, for defining and exchanging QN models between tools; finally, the bridge would also allow to define a behavioral semantics for QNMs, specified by means of a fixed set of behavioral rules in *e-Motions*. Regarding the last point and considering the possibilities and expressive power for modeling DSLs in *e-Motions*, such behavioral semantics can be decorated with new features in order to extend the QNMs' behavior in a more realistic manner. Concretely, we have extended this behavior by considering that servers can fail, and can be temporarily unavailable or out of order.

This chapter reports on the experience on building that bridge, and describes the DSL and the tool (called xQNM) that we have developed using standard MDE techniques. Moreover, and despite xQNM being academic oriented, serving as a proof-of-concept of our approach and being defined at a higher-level of abstraction, we show how it is comparable to other existing QNM tools which were defined using standard programming approaches. In this regard, we also survey many QNM tools from literature. We present the xQNM architecture, based on a layered approach, and the implementation details that contribute to achieve the above mentioned advantages. The complete xQNM toolkit is available from [TV12].



(a) An Open Queuing Network (b) A Closed Queuing Network

Figure 5.1: Examples of an Open and a Closed Queuing Networks.

The rest of the chapter is organized as follows. After this introduction, Section 5.1 presents the background regarding QNMs and several tools. Then, Section 5.2 introduces the abstract syntax of xQNM, in terms of an extension of PMIF 2 [Smi10]. Section 5.3 presents an overview of the components of the xQNM language, describing its semantics in terms of a generic behavioral model for QNMs, its concrete syntax, and the graphical editor we have built to create and input queueing network models. Then, Section 5.4 explains how we deal with QNMs behavioral simulations, it compares them with other tools and presents the extensions needed to consider failures in servers. Finally, Section 5.5 summarizes the chapter.

5.1 Background

5.1.1 Queuing Network Models

In computer systems, many jobs share the system resources such as CPU, disks, and other devices. Since generally only one job (or some of them) can use the resource at any given time, all other jobs wanting to use that resource wait in queues. Systems where jobs may be serviced at one or more queues before leaving the system are modeled with queuing networks. Queuing theory helps in determining the time that jobs spend in various queues in the system [LZGS84]. These times can then be combined to predict the system response time, which is basically the total time that a job spends inside the system, and other non-functional features such as throughput, idle-times, etc.

There are two main types of queuing networks: *open* and *closed*. The former has external arrivals and departures. The jobs enter the system at a source and depart at a sink (Fig. 5.1(a)). The number of jobs in the system varies with time. Closed networks have no external arrivals or departures: the jobs in the system keep circulating from one queue to the next. The total number of jobs in the system is constant. It is possible to view a closed system as a system where the sink is connected back to the source (Fig. 5.1(b)), and jobs leaving the system immediately re-enter it. There are also *mixed* networks, which behave as open for some workloads and closed for others. All jobs of a single class have the same service demands and transition probabilities.

5.1.2 QNM tools

There are several commercial packages to queuing network modeling, like QNAP2 [VP04], the PDQ analyzer [Gun05], SPE·ED [LS 10], RESQME [CGLM93], BEST/1 [BGS77], CSIM [Sch86]. There are also many academic tools including TANGRAM-II [dSeSL00], SHARPE [HSZT00], JINQS [Fie10, HAFK11], qnetworks [Mar10] and JMT [BCS09] (for a very complete list, see [Hly11]).

Table 5.1: Features of some packages and tools for QN modeling and analysis

Tool	Evaluation technique	(Input) Model Format	Probability Distributions admitted	Types of QNM supported
RESQME (1986)	Discrete event simulation	Graphical environment with textual information to draw input models	Erlang, Exponential, Normal, Uniform, etc.	Extended QNMs of resource connection systems
SHARPE (1987)	Analysis	Graphical user interface for drawing input models	It allows s- independent random variables and mixing of distributions. It cannot handle Weibull distributions [ST87]	QNMs and also multiple model types (Fault Tree, Markov Chain, Semi-Markov Chain, MRGP, GSPN, PFQN, MPFQN, Trask graph, etc.)
QNAP2 (1992)	Both discrete event simulation and analysis	Programmatical. The analytical solvers need to be invoked	Erlang, Exponential, Normal, Uniform, etc.	Open, closed and mixed queuing networks
QSIM (1995, Release 6.11)	Discrete event simulation	Graphical user interface for drawing the input models	Exponential, Gamma, Erlang, Uniform, Deterministic, Non-Homogeneous Poisson, etc.	Open and closed networks
SPE·ED (1996)	Analysis and Simulation	Graphical user interface for drawing the input models	Various (for simulation)	Any QNM as well as SPE models as defined in Connie U. Smith's books
PEPSY-QNS (1996)	Both analysis and discrete event simulation	Graphically (with XPEPSY), or textually	Various (for simulation)	Open, closed and mixed networks
TANGRAM-II (1997)	Analysis and simulation	Programmatical (models are composed of objects that interact by exchanging messages)	Exponential, Pareto, Deterministic, Uniform, Erlang, Gaussian, Log-normal, FARIMA, FBM	Models of communication systems (computer networks, traffic systems, etc.)
PDQ (1998)	Analysis	Programmatical (using C)	Exponential distribution	Open and closed networks
MQNA (2003)	Analysis	Textually	Exponential distribution	Open and closed product-form QNs and finite capacity QNs.
WinPEPSY-QNS (2006)	Analysis and simulation (closed queuing systems with capacity and phase type distributions cannot be simulated)	Graphical user interface for drawing input models	Phase-type distributions (approximations of long-tail distributions achieved by finite mixtures of exponentials [FW97])	Stochastic models based on queuing networks with phase-type distributions
JINQS (2006)	Discrete event simulation	Programmatical (in Java)	Exponential, Weibull, Cauchy, Deterministic, Erlang, Gamma, Geometric, Normal, Pareto, Uniform	Any queuing system and queuing network model
JMT (2007)	Analysis and discrete event simulation	Graphical user interface for drawing input models. Wizards are available. It also supports interoperability via XML	Pareto, Gamma, Hyperexponential, Erlang, etc.	Any queuing system and queuing network model
qnetworks (2009)	Analysis	Programmatical (in Octave)	Poisson distributions for arrival rates and Exponential distributions for service times	Open, closed and mixed networks with multiple job classes
xQNM (2012)	Discrete event simulation	Graphical user interface for drawing the input models. Importation of PIMF models is also allowed	Uniform, Exponential, Normal, Gamma, Weibull, Erlang, F, Log-normal, Pareto, Pascal, etc.	Open and closed networks

Table 5.1 presents several relevant features of some of the existing packages and tools for solving QNMs (xQNM has also been included for comparison with the rest). They are listed according to their approximate chronological appea-

rance. For each tool we list the evaluation technique it uses (analytical methods, simulation or both), the specific model representation needed, the probability distributions it accepts and the types of QNMs it can analyze. Most of these tools were developed some years ago, and each of them specifies a queuing network model in a different way and with a different language. To address the problem of exchanging models among tools, a performance model interchange format (PMIF) was proposed [Smi10, Smi04, GLSP06, SW99]. PMIF provides a common representation for system performance model data that can be used to exchange models among QNM modeling tools. However, still most of the existing tools are not able to receive a PMIF model as input. It is true that some tools tried to define common formats for tool interoperability purposes, with goal similar to PMIF. This is the case of MOSEL-2 [WdMBB06], a tool that provides means for specifying QNMs and carrying out some performance measurements over them. The tool is equipped with a set of model translators that allow the automatic transformation of MOSEL-2 models to several third-party performance evaluation tools. WEASEL [SEA12] is an interesting client-server application in which the user can specify a PMIF 2 (see Sect. 5.1.3) model graphically and then solve it by using the following external solution tools: PDQ, SHARPE, MVACCKSW (MVA using different methods) and PEPSY. Furthermore, it offers the option to translate the PMIF 2 model to the specific notation of different tools, such as PDQ, SHARPE, PMVA, QNAP, OPENQN, CLOSEDQN, MVAQFP, MQNA1, MQNA2 and PEPSY.

Only some of the tools mentioned provide a graphical interface for the definition of QNMs (namely RESQME, SHARPE, SPE·ED, PEPSY, JMT, QSIM and xQNM), in the rest the input models have to be introduced textually or programmatically. And in most cases, all these formats are proprietary and cannot be easily ported to other tools.

Analytical methods do not allow the exact evaluation of the performance of QNMs with arbitrary probability distributions for arrival and service times, only if they use Exponential and Uniform distributions. This is why many packages also offer solutions based on simulation for dealing with other distributions: TANGRAM-II, SPE·ED, QNAP2, WinPEPSY-QNS and the JMT suite. Our tool belongs to this group.

Among the tools described in Table 5.1, there are tools written in FORTRAN (QNAP2), C++ (TANGRAM-II and WinPEPSY-QNS), C (PEPSY-QNS, PDQ Analyzer), GNU Octave (qnetworks) and Java (JINQS, JMT). This is one aspect in which our tool significantly differs from the rest, because it has been developed using MDE techniques, and is defined in terms of DSLs and model transformations between them, at a higher level of abstraction. This allows us the possibility to modify or improve one of its parts and keep the rest untouched, and provides us with a very organized and modular architecture. Consequently, it makes the tool easier extensible for future versions and improves its maintainability. jEQN [GD07] is a DSL for the specification and implementation of distributed

simulators for extended queueing networks. Although it also uses MDE techniques and provides a DSL for specification and simulation, it builds on Java while our approach relies on an existing DSL for the specification of real-time systems. Besides, jEQN focuses on the development of distributed simulators from local ones for extended QNMs while our tool focuses on the definition and management of QNMs (definition, importation, exportation) as well as on their simulation.

Most of the works about QNMs do not consider failures. This is, the servers that compose the network can fail, being unable to process jobs for some time and contributing to system delay. In this sense, these works consider that the networks have an "ideal" behavior, where nothing can go wrong. But this is far from reality, since in many systems modeled with queuing networks many things can go wrong. For example, in manufacturing systems, the machines that make up the system can fail, or the actual servers that compose any kind of network modeled with a QNM can have failures too (hardware failures, failures due to wear out, random failures, etc.).

There are some works that do take into account failures of this type. For example, Das and Murray Woodside [DMW98] consider that any of the entities in a model can undergo a failure, which is independent of the failures of other entities in the model. Each entity i has its own component state, S_i (0 or 1), corresponding to its working state or failed state, and is governed by a separate Markov chain with a working state ($s_i = 1$) and a failed state ($s_i = 0$), with rates of failure and repair. We have applied this idea of networks' components having two states to extend the behavior of ordinary QNMs (see Section 5.4.4). Altiok [Alt97] has reviewed in detail literature pertaining to queues with service breakdowns due to failures of service stations. S. Kumar and P. R. Kumar consider machine's failures in manufacturing systems [KK94], and assign exponential times for times to failure and times to repair. Govil and Fu survey in [GF99] contributions and applications of queuing theory in the field of discrete part manufacturing, where they reference other works dealing with failures in manufacturing flow lines [Kei62, FG86, Alt89].

5.1.3 Evolution of PMIF

PMIF was conceived as a common representation for system performance model data that could be used to move models between modeling tools [SW99]. Its creators were interested in tool interoperability for *Software Performance Engineering* [Smi90]. Its structure represents the software processing steps and other information for workloads that execute in the system performance level. PMIF, however, was born for system performance models that represent computer platforms and network interconnections with a network of queues and servers. Its representation technique had to be appropriate to express the interchange format and it needed to be capable of expressing a wide range of system execution models: those containing a small number of servers to a very large number of them, from one to many workloads, both open and closed models, that may be solved using

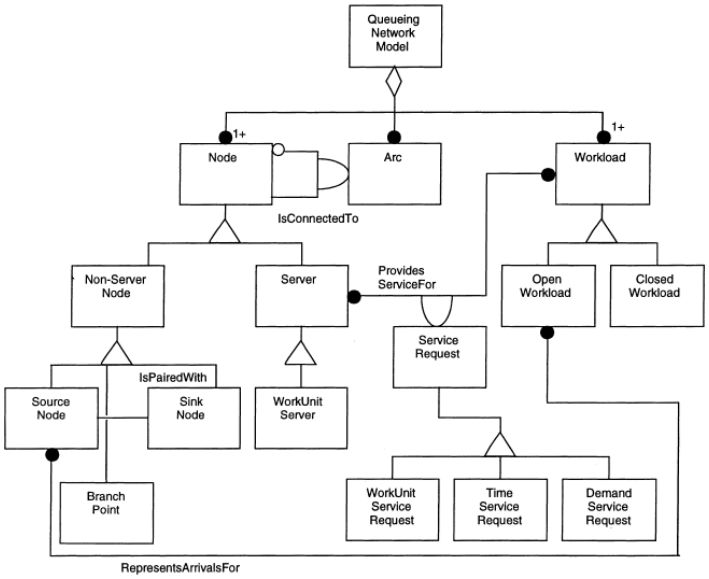


Figure 5.2: PMIF 1.0 Metamodel

either analytical or simulation solution techniques. It also had to be usable with existing tools, include modeling features that tools provide, support the modeling paradigms prevalent in tools, and use terminology common in tools and modeling research. So the first version of PMIF (1998), as explained in [SW99], addressed a specific type of performance model: Queuing Network Models that may be solved using exact analytical solution algorithms. The resulting metamodel is shown in Figure 5.2. In this version, the use of the operational analysis term *visits* rather than the stochastic modeling *probability* among servers was proposed.

A new version of the PMIF metamodel and its XML schema specification (called PMIF 2.0, and later PMIF 2) was then presented in [Smi04, SL04, Smi10]. An XML-based approach was used to tackle the complexity and amount of effort required to create the PMIF interface. It uses the previous PMIF (PMIF 1.0) metamodel as a starting point because it is a good description of the information requirements for performance model interchange, but uses XML to implement the transfer format. As previously mentioned, the PMIF 1.0 metamodel uses number of visits instead of routing probabilities, assuming that from the number of visits, and with the knowledge of the queuing network topology, routing probabilities can be calculated. This assumption is true for many of the queuing networks that model computer systems. However, it is not true for the general case. This is why the routing probability was added as a transit element which specifies where a job has to transit and with what probability. One of the advantages of PMIF is that it can be used by web services to export and import QNMs among different modeling tools. In [RLPS05], PMIF 2 is used as the exchange format of QNMs among SPE·ED and QNAP by means of a web service. First, the software model

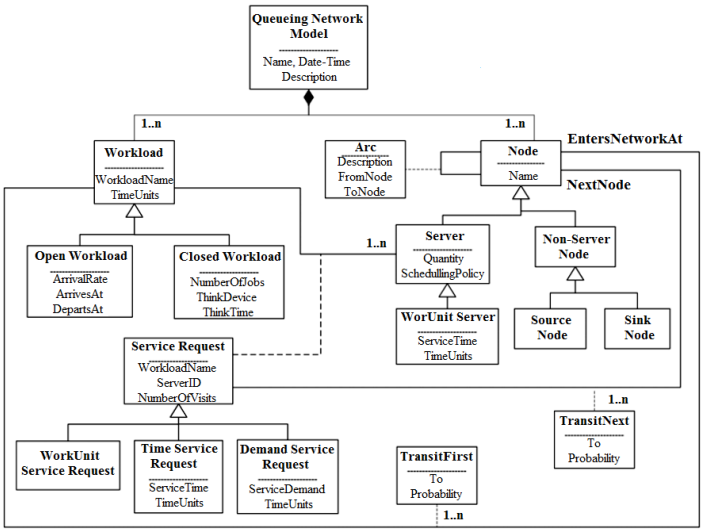


Figure 5.3: PMIF 2 Metamodel (from [Smi10])

created in the SPE·ED performance modeling tool is exported to the PMIF 2 format. Then, it is transformed to the QNAP notation, after which the model is ready to be analyzed by QNAP. The PMIF 2 metamodel is shown in Fig. 5.3.

In this chapter, we take a step forward because our aim is not just to be able to describe models in XML, but to integrate them into the MDE tool chain. Thus, we have used Ecore [BMS06] as meta-metamodel (shown in Appendix C), and so Ecore models representing queuing network models expressed in PMIF can be defined using Eclipse. Furthermore, several probability distributions for arrival and service times can be specified in the models. This is further explained in next section.

5.2 Expressing PMIF in Ecore

The metamodel conforming to Ecore [BMS06] that we propose for defining QNMs, named ePMIF (for *Ecore*-PMIF), is shown in Figure 5.4. It can be seen as the MDE version of the PMIF 2 metamodel presented in [Smi10] (Fig. 5.3), with some minor changes.

A QueuingNetworkModel is composed of one or more Workloads, zero or more Arcs, one or more Nodes and one or more ServiceRequests. The Arc class connects Nodes between them. In a queuing network, jobs flow from node to node. There are two types of nodes, Servers and NonServerNodes. The former provide a processing service, while the latter show the topology of the network and represent the origin (SourceNode) and exit point (SinkNode) of OpenWorkloads. The Server class has a specialization class, named WorkUnitServer, that represents resources with a fixed processing service for each Workload that makes a request

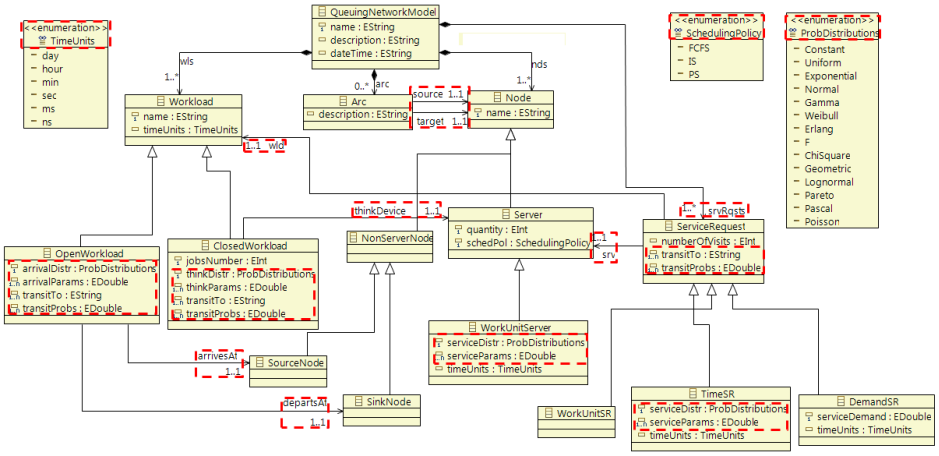


Figure 5.4: ePMIF metamodel (conforming to Ecore)

for service.

A Server provides service for different Workloads, where a Workload represents a collection of jobs that make similar ServiceRequests from Servers. Depending on the type of queuing network (open or closed), there are two types of Workloads:

- OpenWorkload. It represents a set of jobs which arrive from the outside world, are serviced, and leave the system. The number of jobs belonging to an OpenWorkload at any given time is variable. A job represented by an OpenWorkload arrives at a SourceNode and departs at a SinkNode.
- ClosedWorkload. It represents a fixed population of jobs that circulates among the Servers. A ClosedWorkload has a Server which acts as thinkDevice and which is characterized by a think time.

A ServiceRequest associates Workloads with Servers. According to the relation from ServiceRequest to Workload and Server (one to one in both cases), a ServiceRequest associates one (and only one) Server with one (and only one) Workload. In this way, when a job which is represented by a workload arrives at a server, the service request associated to the workload and the server specifies how the job will be treated in that server. There are three types of ServiceRequests (for all of them, the numberOfVisits is an optional attribute):

- WorkUnitServiceRequest. They are ServiceRequests associated to WorkUnitServers, so nothing about the service time has to be specified, since it is already in the WorkUnitServer.
- TimeServiceRequest. It specifies the service time that the jobs representing the Workload associated to the ServiceRequest will have in the associated Server.

- DemandServiceRequest. Similar to TimeServiceRequest, but service time is now specified in terms of service demand (service time multiplied by number of visits).

All these elements are equivalent to those in PMIF 2 (Fig. 5.3), apart from the following differences (they are marked with dotted boxes in Fig. 5.4). First, in PMIF 2, probabilities are specified as classes in the metamodel, while in our approach they have become attributes (to reduce the number of elements in the resulting models, mainly for simulation performance reasons). Second, ServiceRequest is no longer an association class, and we have also unified the way to specify transitions in Workload and ServiceRequest classes (this will be very useful when specifying the behavior). Thus, in the PMIF 2 metamodel an element of type Transit was needed for each path in a fork; in our case, no matter how many paths a Workload may follow, we only need two attributes: transitTo and transitProbs. The former contains a sequence with the names of the Nodes where the Workload can transit. The latter contains a sequence with the probabilities of these transitions. Note that the order of the elements in both attributes has to match.

For example, suppose that in the network shown in Fig. 5.1(a), the probability of a job to transit from the CPU server to DISKA is 0.4375, to DISKB is 0.5, and to leave the system is 0.0625 (example taken from [Jai91, page 572]). This is represented in our approach by setting the values of attributes transitTo and transitProbs of the ServiceRequest associated to the CPU server, to the sequences {DISKA, DISKB, SINK} and {0.4375, 0.5, 0.0625} respectively.

Another difference between ePMIF and PMIF 2 is how service and arrival times are specified. In PMIF 2, they are specified by attributes ArrivalRate and ServiceTime respectively. PMIF 2 assumes that these values represent parameters of Poisson and Exponential distributions, respectively. Given that we want to accept different probability distributions for service and arrival times, we have defined a new type (ProbDistributions) which is an enumeration with literals Constant, Uniform, Exponential, Normal, Gamma, Weibull, Erlang, F, ChiSquare, Geometric, Lognormal, Pareto, Pascal and Poisson. If the distribution is Constant, it means that the arrival/service time is constant.

The last difference between PMIF 2 and ePMIF is that we use references instead of attributes to refer to other objects. This has the advantage that references cannot be incorrectly specified. However, if objects are referred to by their names, it is easy to mistakenly write a String with a name that refers to a non-existent object. Furthermore, a change in the name of an object would result in an inconsistent reference.

5.3 xQNM overview

Fig. 5.5 shows the basic elements of xQNM, and how to use them to specify and simulate QNMs.

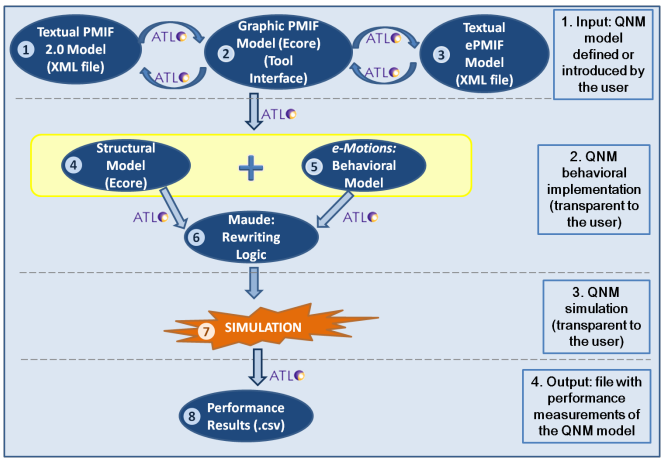


Figure 5.5: xQNM Architecture

The concrete syntax of the language and the tool support we have built for editing xQNM models (that is, QNMs) is described first in Section 5.3.1. It corresponds to the ovals numbered 1, 2 and 3 in the figure. Section 5.3.2 explains how the behavior of queuing network models can be specified with *e-Motions*. It provides the *semantics* of the xQNM language, and corresponds to the ovals 4 and 5 in the figure.

Once we have an initial model of a QNM and the behavioral dynamics of QNMs specified in *e-Motions*, we can simulate it. In fact, *e-Motions* translates its specifications (using ATL transformations from ovals 4 and 5 to oval 6 in Fig. 5.5) into the corresponding formal specifications in Real-Time Maude [CDE+07], which in turn provides semantics to the visual *e-Motions* specifications of the system. Maude specifications are executable, and therefore they can be used to run simulations. Section 5.4 describes the simulations that are possible with xQNM, how they are realized, and how results are returned to the user.

5.3.1 A Tool for Drawing and Simulating QNMs

Our DSL is supported by a tool which provides a graphical editor for creating queueing networks conforming to PMIF or ePMIF metamodels. It means that it can be defined open, closed and mixed network models in the graphical interface. At this moment, only open and closed networks can be simulated in xQNM. This section explains the capabilities provided by this tool.

QNMs graphical definition

The graphical editor of our tool has been developed using GMF. Fig. 5.6 contains a snapshot of our editor, with the graphical representation of the QNM model showed in Fig. 5.1(a). The different kinds of network objects (OpenWorkloads,

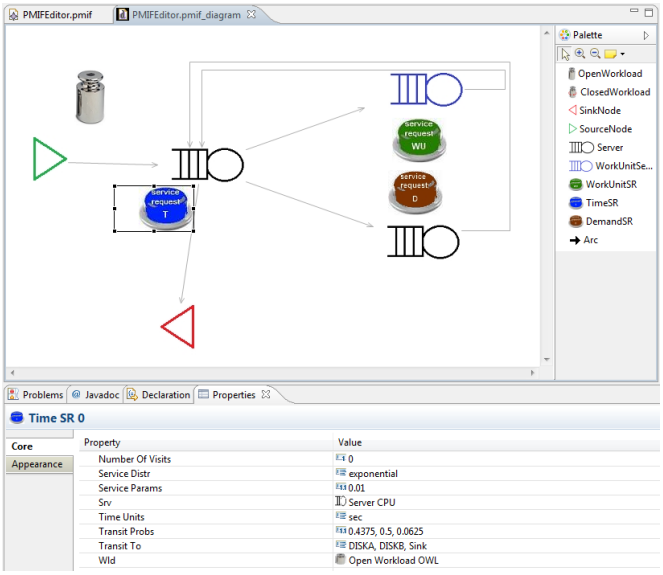


Figure 5.6: xQNM Graphical Editor

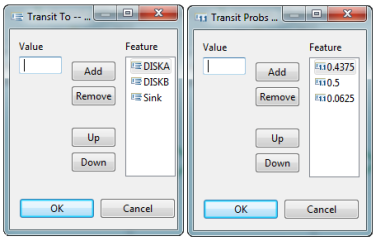
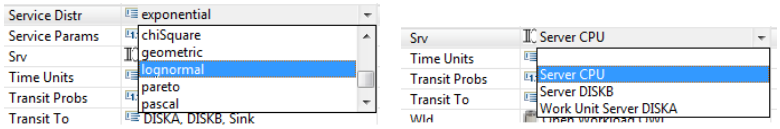


Figure 5.7: Assigning values to sequences

ClosedWorkloads, Servers, WorkUnitServers, etc.) can be selected from the menu on the right and be placed on the main panel.

The properties of objects (attributes and references) are specified in the lower panel. To assign values to the sequences of transitions, the user has to select the object and click on the attribute in the lower panel. A new window where the values can be introduced is shown in Fig. 5.7. Probability distributions are specified as attributes of type ProbDistribution (Fig. 5.8(a)). References to objects (that model for example transitions) are indicated using drop down lists (Fig. 5.8(b)).

As in any GMF project, xQNM models admit two representations, each one stored in a different file. One contains the graphical information, and can be edited with our graphical tool. The second one is plain XML file that contains the model elements, and can be edited with the standard Eclipse tree-view model editor. The user can select either of them in the left panel.



(a) Specifying the distribution.

(b) Selecting the Server associated to the ServiceRequest.

Figure 5.8: Drop down lists in the lower panel.

Exporting QNMs

Once a queuing network model is defined with the graphical editor, it can be exported to an XML file with its ePMIF representation. The XML is similar to the PMIF 2 XML file, with the corresponding extensions for transitions and probability distributions. Thus, there are no ArrivalRate, ServiceTime and ThinkTime attributes anymore; but ArrivalDistr, ServiceDistr and ThinkDistr. Objects containing any of these attributes also contain one or more attributes named Param that specify the parameters of the distributions. For instance, let us consider the example shown in Fig. 5.1(a) and described in Section 5.2 of an open QNM with a CPU and two disks: A and B. Distributions for service times are supposed to be Gamma (for disk A) and Exponential (for disk B), and Poisson for arrival times. Listing 5.1 shows the XML file that has been exported from the definition of this open network model using our tool.

Listing 5.1: ePMIF XML File

```
<QueueingNetworkModel Name="Jain572" Description=
 "Ecore XML PMIF" Date-Time="040711">
  <Workload>
    <OpenWorkload WorkloadName="OWL" ArrivesAt="Source"
     DepartsAt="Sink" ArrivalDistr="Poisson"
     TimeUnits="sec">
       <Transit Probability="1.0" To="CPU"/>
       <Param Value="3.0"/>
    </OpenWorkload>
  </Workload>
  <Node>
    <Server Name="CPU" Quantity="1"
     SchedulingPolicy="FCFS"/>
    <Server Name="DISKB" Quantity="1"
     SchedulingPolicy="FCFS"/>
    <WorkUnitServer Name="DISKA" Quantity="1"
     SchedulingPolicy="FCFS" TimeUnits="sec"
     ServiceDistr="Gamma">
       <Param Value="0.5"/>
       <Param Value="2.0"/>
    </WorkUnitServer>
    <SourceNode Name="Source"/>
    <SinkNode Name="Sink"/>
  </Node>
  <ServiceRequest>
    <DemandServiceRequest ServiceDemand="2592.0"
     TimeUnits="sec" WorkloadName="OWL"
     ServerID="DISKB" NumberOfVisits="86400">
```

```
      <Transit Probability="1.0" To="CPU"/>
    </DemandServiceRequest>
    <WorkUnitServiceRequest WorkloadName="OWL"
     ServerID="DISKA">
      <Transit Probability="1.0" To="CPU"/>
    </WorkUnitServiceRequest>
    <TimeServiceRequest TimeUnits="sec"
     WorkloadName="OWL" ServerID="CPU"
     ServiceDistr="Exponential">
      <Param Value="0.01"/>
      <Transit Probability="0.4375" To="DISKA"/>
      <Transit Probability="0.5" To="DISKB"/>
      <Transit Probability="0.0625" To="Sink"/>
    </TimeServiceRequest>
  </ServiceRequest>
  <Arc FromNode="Source" ToNode="CPU"/>
  <Arc FromNode="CPU" ToNode="DISKA"/>
  <Arc FromNode="CPU" ToNode="DISKB"/>
  <Arc FromNode="CPU" ToNode="Sink"/>
  <Arc FromNode="DISKA" ToNode="CPU"/>
  <Arc FromNode="DISKB" ToNode="CPU"/>
</QueueingNetworkModel>
```

Our tool also supports the exportation to standard PMIF 2 XML format, as long as the distributions are those supported by PMIF 2.

Importing QNMs

The xQNM tool offers the possibility to import PMIF 2 files. These files are transformed into the corresponding ePMIF files, modifying the attributes as required. Thus, attributes ServiceTime and ThinkTime are automatically translated into the corresponding serviceDistr and thinkDistr attributes. New attributes serviceParams and thinkParams are created with the values of ServiceTime and ThinkTime attributes, respectively. The same happens with the ArrivalRate attribute in PMIF 2, which is translated to an arrivalDistr attribute (of type Poisson in this case).

It is also possible to import plain ePMIF XML files to the xQNM tool (i.e., with no graphical information). The ATL transformation used for this is the opposite to the one used for the exportation of ePMIF models, explained above.

When plain XML files containing either PMIF or ePMIF models are imported into our tool, a file containing the new model generated is created. Having no graphical information about the model, this file only accepts the visualization using the Eclipse tree-view editor. From this file, the user can generate another file so that the model can be deployed in the graphical editor. The elements will initially appear in a random position in the graphical editor (since no graphical information is available), and the user is then free to arrange them as preferred.

5.3.2 A Generic Behavioral Model for QNMs

The *generic* behavioral model for open and closed QNMs is defined in terms of a set of *e-Motions* rules. Note that users of the xQNM tool do not need to be

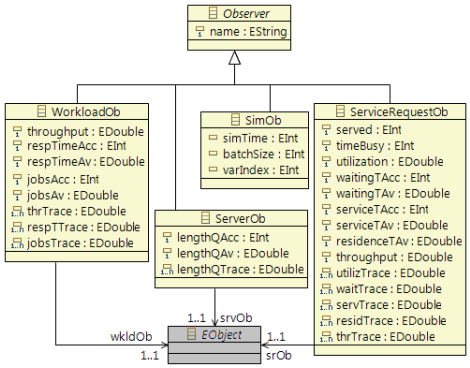


Figure 5.9: *Observers* metamodel

aware of such behavioral model. Mixed queuing network models defined using the graphical user interface of xQNM cannot be simulated at this moment.

QNMs structural model and Observers addition

PMIF models describe the structure of the QNM, and can be used to specify the dynamics of QNMs in terms of job flows. However, we also need to specify, record and manage additional information to deal with the performance properties of the system. For these tasks we use *observers*. Observers were introduced in Chapter 3 as an effective means to specify the non-functional properties of systems described by high-level DSLs. An observer is an object whose purpose is to monitor the state of the system objects and actions. Observers, as any other objects, have a state and a well-defined behavior. The attributes of the observers capture their state, and are used to store the variables that we want to monitor.

To introduce observers into the behavioral rules of xQNM (in order to specify and measure the performance properties of QNMs), we need to specify a metamodel for them. This is shown in Fig. 5.9. The idea is to combine both metamodels (Figs. 5.4 and 5.9) so that observers can be used in our behavioral rules. In fact, since *e-Motions* allows users to merge several metamodels in the definition of a DSL, we can define the observers metamodel in a non-intrusive way, i.e., we do not need to modify the system metamodel to add attributes that store the values of the non-functional properties we want to monitor.

In the observers metamodel we can see that there are three types of observers for monitoring the performance properties of a different type of object. We have WorkloadOb for monitoring Workloads, ServerOb to monitor Servers, and ServiceRequestOb to monitor TServiceRequests. These three have a reference to class EOject, which points to the object they monitor. In addition, we have the SimOb observer, which stores the simulation run parameters introduced by the user (see Section 5.4.1).

The aim of WorkloadOb observers is to monitor performance properties of

workloads. The idea is to associate one observer of this type to each workload. Its attributes are used to measure the average throughput (thoughputAv), response time (respTimeAv) and jobs (jobsAv) of the associated workload. It also contains three sequences (thrTrace, respTTrace and jobsTrace) that store the traces with the values for throughput, response time and jobs average, respectively, at different times of the simulation.

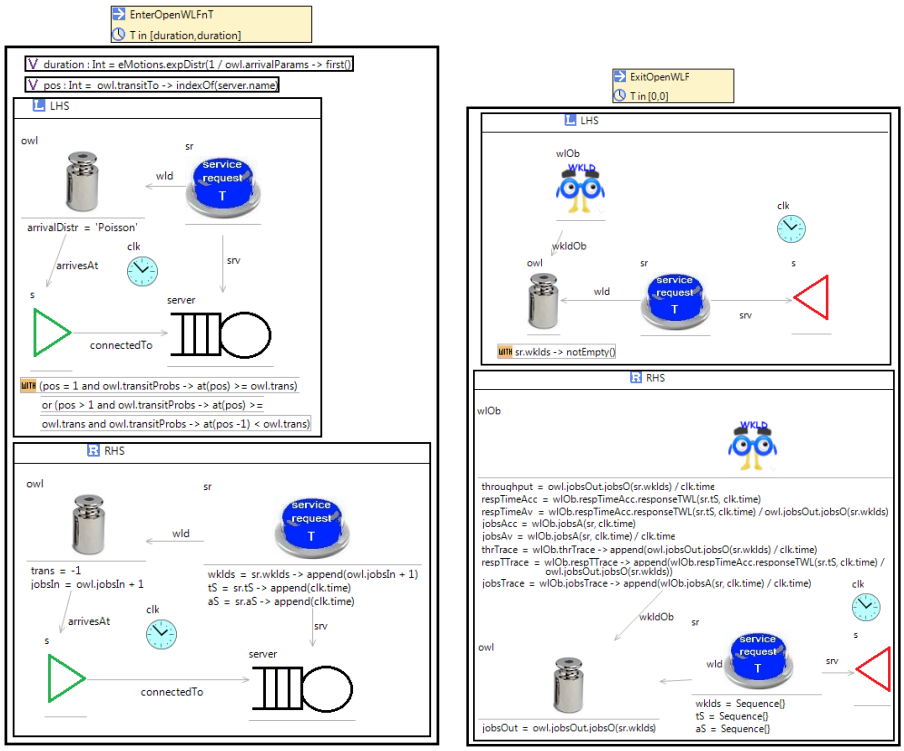
ServerOb observers monitor servers. They store the average queue length in their attribute lengthQAv, and keep the traces in attribute lengthQTrace. Attribute lengthQAcc is used to compute lengthQAv. As explained in [Jai91], the queue length of a server considers the jobs in the queue and the jobs being served.

Each service request in the model will have a ServiceRequestOb observer associated to it. Considering that a service request is the relationship between a server and a workload that requests its service, the data monitored by this observer represents the performance relationship between them. In this way, when we mention workloads (or jobs belonging to them) and servers in the explanation of the attributes, we mean the workloads (or jobs) and servers associated to the service request. ServiceRequestOb observers have several attributes:

- served. Number of jobs processed by the server.
- timeBusy. Time that the server has been busy (processing jobs).
- utilization. Percentage of the time that the server has been busy.
- waitingTAcc and waitingTAv. Sum and average waiting times in the queue of the jobs processed by the server, respectively (the waiting time of a job is the time between the arrival of the job to the server queue until it starts being processed).
- serviceTAcc and serviceTAv. Sum and average service time of all jobs processed by the server.
- residenceTAv. Average residence time of all jobs processed by the server (the residence time of a job is the time between the job enters the server queue and leaves the server).
- throughput. Number of jobs processed by the server per unit of time.
- utilizTrace, waitTrace, servTrace, residTrace and thrTrace. These attributes keep the traces of the corresponding values throughout the simulation.

QNMs behavioral model

This section introduces the *e-Motions* rules that describe the behavior of QNMs. Basically there is one rule for jobs entering the network (EnterOpenWLFnT), one for specifying how jobs transit between servers (TransitJobsnT), and a third



(a) EnterOpenWLFnT rule (b) ExitOpenWLF Rule

Figure 5.10: Rules for packets entry and leaving.

one for jobs leaving the network (ExitOpenWLF). For efficiency reasons there are variations of these rules when there is only one server to which the jobs can transit to (so no decisions are to be made). In addition, two rules are in charge of specifying how the values of global observers are updated.

These rules are briefly described here. For a complete description of all the rules, the interested reader can consult [TV12]. In any case, the rules are completely transparent to the xQNM user, they just specify the behavior of the system, and allow to simulate it.

a) A set of jobs enter the network. Rule EnterOpenWLFnT (Fig. 5.10(a)) models how OpenWorkload objects enter the network, when they can transit to more than one server. The rule has in both LHS and RHS patterns the OpenWorkload to which the job belongs (owl), the Server to which the job transits to (server), the ServiceRequest that relates both of them (sr), and the Source node at which jobs belonging to the OpenWorkload enter (s). There are also the relationships between these objects (wld, arrivesAt, srv and connectedTo).

The destination server is determined by variable pos and the OCL condition in the LHS. It uses the transition probabilities. A new job entering the system is modeled by the addition of a new identifier to the wklds sequence of the

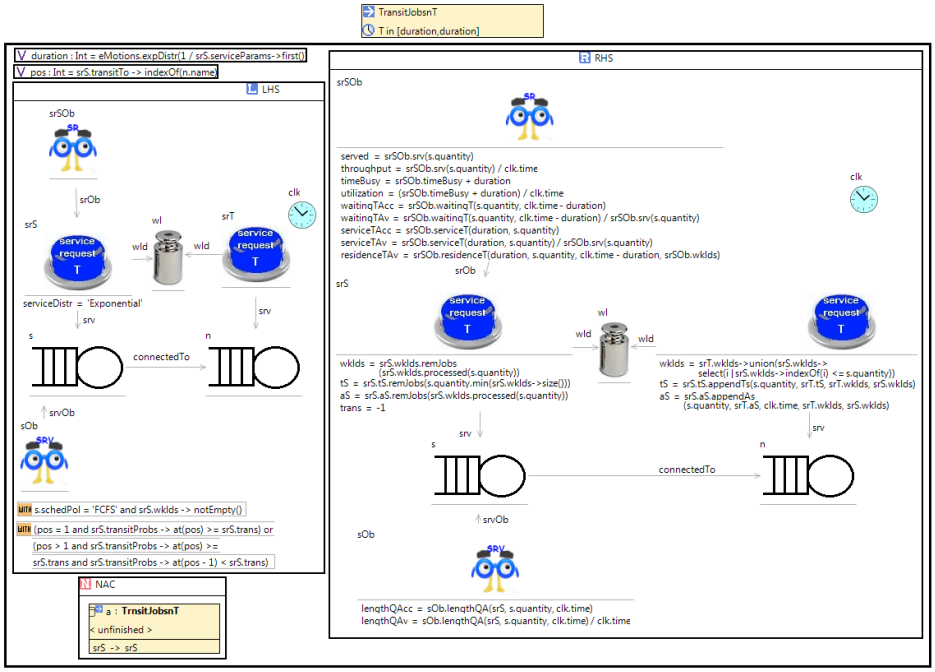


Figure 5.11: TransitJobsnT rule

TServiceRequest, and the addition of the current time elapse to the tS and aS sequences. Variable duration specifies the duration of the rule: in this example it follows a Poisson distribution (see the variable duration declaration in the top left corner of the rule).

There is also a similar rule for OpenWorkloads whose jobs always transit to the same Server when they enter the Source node. That rule, called EnterOpenWLF1T [TV12], is a simplified version of the EnterOpenWLFnT rule that we have developed for performance reasons (because no OCL expressions or conditions need to be computed in this case).

b) Transition of jobs between servers. Rule TransitJobsnT (Fig. 5.11) models the transition of jobs between servers (and also from a Server to a Node of type SinkNode). Jobs can belong to either OpenWorkloads or ClosedWorkloads, so this rule is used for both.

The LHS of rule TransitJobsnT contains all the objects needed for this rule to be triggered: the source Server (s), the target Node (n, which is either a Server or a SinkNode), the Workload (wl) to which jobs belong, the TServiceRequests associated to the mentioned elements (srS and srT), and the Observers (srSOb and sOb) whose attributes are to be updated in the RHS of the rule. The three sequences representing the jobs in the source TServiceRequest (srS) are also updated in the RHS by eliminating the corresponding jobs and adding them to the sequences of the target TServiceRequest (srT). The attributes of the two

observers are also updated.

c) Jobs leave the network. Rule ExitOpenWLF (Fig. 5.10(b)) models how jobs leave the network. Consequently, it is applied only over OpenWorkloads. When the TServiceRequest (sr) contains jobs, this rule is fired and the corresponding attributes in the OpenWorkload (owl) and the observer associated to it (wlOb) are updated. The jobs present in the TServiceRequest (sr) are deleted, modeling that they have left the network.

This rule updates the attributes of the observers, namely thrTrace, RespTTrace and jobsTrace, every time a job leaves the system. The new values correspond to the calculated throughput, mean response time and jobs average, which are appended to the sequences with the traces.

Similar to rule ExitOpenWLF, another rule is in charge of updating the attributes of observers associated to ClosedWorkloads. The attributes are the same, apart from the one for the average number of jobs, which is no longer necessary.

Finally, another rule, UpdateTraces (not shown here for brevity), is defined to update the attributes for traces in the other observers. They are updated either when jobs leave the system (in OpenWorkloads) or when jobs arrive at the centralSrv (for ClosedWorkloads).

It is important to recall that users do not need to write these rules, they have been defined once and apply to all QNMs. In fact, they can be seen as providing a behavioral semantics of QNMs by explicitly specifying the behavior of QNMs in a language with well-defined semantics [RDV10]. In addition, they are all automatically configured and generated according to the type of network defined by the user, and to the probability distributions used.

Generating the behavioral rules

Once the user inserts a model within the xQNM tool either by drawing it with the graphical interface or by importing it, it is automatically translated to its structural and behavioral models. This is done by the ATL transformation shown in Fig. 5.5 from oval 2 to ovals 4 and 5.

The transformation has two main parts, the generation of the structural model and the generation of the behavioral model. For the former, the transformation takes the ePMIF model conforming to the ePMIF metamodel and transforms it into an more compact representation of ePMIF that we use internally with *e-Motions*. Although the first version of xQNM used ePMIF directly, we realized that for performance reasons we could optimize this representation to make it more compact and efficient. This was very important for conducting the simulations. Such new representation is internal to our tool and transparent to users, who still use ePMIF models to describe their QNM models. That metamodel and the changes with respect to ePMIF are described in detail in [TV12] and Appendix C.

For generating the behavioral model, the transformation identifies the type of queuing network used (open or closed) and selects the appropriate rules among the ones presented in Section 5.3.2, which are available in a repository. Those rules, as

well as the ePMIF model are the input parameters of the ATL transformation. The transformation also adjusts some features of the rules according to the probability distributions used in the model, which is reflected in the rules duration, or the performance properties that the user wants to monitor. Regarding the latter, only those metrics are filtered by the ATL transformation and appear as objects attributes in the final rules (Figure 5.12 shows how such parameters are specified by the user). It means that simulations where less parameters are to be monitored are faster.

5.4 Experimentation

5.4.1 Simulation in xQNM

Once we have the behavioral dynamics of a QNM specified in *e-Motions*, we are ready to simulate it. Our environment supports the translation of the specifications (ATL transformations from ovals 4 and 5 to oval 6 in Fig. 5.5) into the corresponding formal specifications in Real-Time Maude [CDE+07].

In Maude, the result of a simulation is the final configuration of objects reached after completing the rewriting steps, which is nothing but a model. The semantic mapping as well as the transformation process back and forth between the *e-Motions* and Real-Time Maude specifications is described in detail in [RVD09], although it is completely transparent to the *e-Motions* (and so xQNM) user. The user, consequently, is completely unaware of the Maude rewriting engine performing the simulation.

A very important advantage of our approach is that observers are also objects of the system, and therefore we can retrieve the values of their attributes after the simulation is conducted to know how the system behaved. In fact, this is crucial for the approach we are presenting here.

When the user wants to launch a simulation in xQNM, the window shown in Figure 5.12 is displayed. Users have to specify the input queuing network model to be simulated. Moreover, they have to specify which performance measures want to obtain as output for each service request, server and workload. Since the behavioral rules presented in Section 5.3.2 are available in a repository, the ATL transformation from oval 5 to oval 6 filters only those attributes that the user wants to monitor. Besides, depending if the network model is open or closed, the transformation filters the appropriate rules to be used in the simulation. Finally, the stopping criteria has to be determined. It can be established either by the desired simulation time or the method of the batch means. The settings for the stopping criteria are stored in the simOb observer (Figure 5.9).

As mentioned in Section 3.5.2, an important issue of any simulation in any kind of system is the stopping criteria. The simulation should stop at a certain point where the performance parameters are accurate enough, and the system is stable. To be able to reach that point, an important requirement is Little's Law [Jai91],

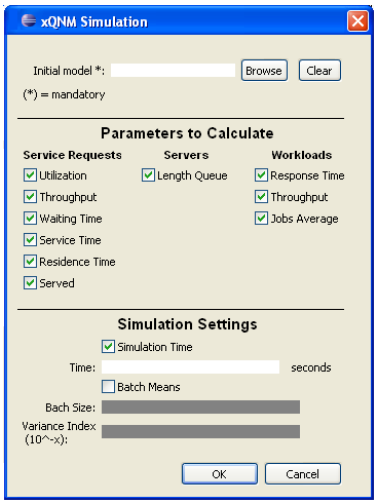


Figure 5.12: xQNM Simulation Window

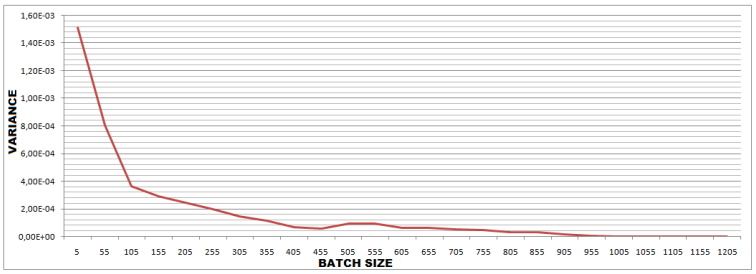


Figure 5.13: Method of the Batch Means

which in a queueing network applies as long as, in average, the number of jobs entering the system are less or equal to those leaving it. It also implies that the average arrival rate of jobs should be lower than the service time, if we do not want the network to overflow.

Related to the stopping criteria of a simulation, we also need to determine its length, that is, for how long it should run. Thus, if the simulation is too short, the results will probably be unreliable. But if the simulation is too long, computing time and resources will be unnecessarily wasted. In most simulations, only the performance after the system reaches a stable state is of interest. The initial part, also called transient state or warm-up period, should not be included in the final computations. The problem of identifying the end of the transient state is termed as transient removal. Some of the common heuristic methods for transient removal are: long runs, proper initialization, truncation, initial data deletion, moving average of independent replications and batch means.

In our approach, we allow *long runs* and the *batch means* methods. We discarded the *proper initialization* method because it requires starting the simulation in

SERVICE REQUESTS

ServiceRequest	owl, CPU
Utilization	0,47365
Throughput	47,46225
Waiting Time	0,00823
Service Time	0,00992
Residence Time	0,01815
Served	38755

ServiceRequest	owl, DISKA
Utilization	0,41726
Throughput	20,76636
Waiting Time	0,01347
Service Time	0,02012
Residence Time	0,03359
Served	16850

ServiceRequest	owl, DISKB
Utilization	0,71737
Throughput	23,83625
Waiting Time	0,07586
Service Time	0,03017
Residence Time	0,10603
Served	19405

SERVERS

Server	CPU
Length queue	1,62958

Server	DISKA
Length queue	1,48269

Server	DISKB
Length queue	2,86924

WORKLOADS

OpenWorkLoad	owl
Response Time	1,37598
Throughput	3,06168
Jobs Average	5,96572

Figure 5.14: Results obtained for the QNM of Fig. 5.1(a)

a state close to the expected steady state. We cannot follow this approach because in principle we do not know the expected steady state. The *truncation* method is based on the assumption that the variability during the steady state is less than during the transient state, which is normally true. It considers that the data in the transient state is monotonous, i.e., continuously increasing or decreasing. However, we found out that our simulations of QNMs may have significant peaks during their transient states. Methods *initial data deletion* and *moving average of independent replications* require studying the overall average after some of the initial observations are deleted from the sample. These methods apply over several replications, which differ only in the seed values used in the random number generators, of a fixed size. The problem, again, is how to determine a priori the length of the replications. This makes them unappropriate for our proposal.

To determine when a simulation has to stop, the original method of *batch means* requires running a long simulation and later dividing it up into several parts of equal duration, which are called batches. The mean of the observations within each batch is called the batch mean. This method requires studying the variance of these batch means as a function of the batch size. But instead of running a very long simulation and later dividing it, what we do is to apply the method at certain points during the simulation as it moves forward. For that we store the values of the performance parameters in traces, and apply this method over them every time that N new jobs leave the system (or, in the case of closed networks, when N jobs complete a cycle). We consider that a simulation has reached the steady state when the variance of the batch means is in the order of 10^{-x}.

Fig. 5.13 shows a chart with the variance (Y-axis) plotted as a function of batch sizes (X-axis), using $N = 50$ and $x = 6$. In this example, the variance goes below 10^{-6} between steps 905 and 955, i.e, when 955 jobs have been processed. This

means that the steady state of the system starts when 955 jobs have left the system (or completed a cycle in a closed network). Based on the experiments we have conducted with different networks, the default values we have currently assigned to these two variables are $N = 100$ and $x = 6$. Of course, these parameters can be easily configured by the user (Figure 5.12).

We run independent simulations, each one being stopped when it reaches its steady state as detected by the batch method. The performance values returned by each simulation are the values of the attributes of the observers defined for the model, at the end of the simulation. Given that we have reached the steady state, the values are stable. To compute the final result we take the average of these values, and the result is returned to the user.

When all the simulations have stopped and the performance results are ready to be returned to the user, a final ATL transformation is applied to the results. This transformation is shown between ovals 7 and 8 in Fig. 5.5. Its goal is to return the data in a format that can be easily consulted, managed and manipulated by the user. This is why we have chosen the *csv* (comma-separated values) format, which is readable by most spreadsheet applications. Figure 5.14 shows the results returned by our approach for our open queue network example.

5.4.2 Simulation in other tools

Table 5.2 displays the relevant features of some packages and tools regarding analysis and simulation of QNMs. For each one we explain how the performance results are shown to the user, the stopping criteria used by the tool (for tools that perform simulations) and the accuracy or confidence interval of the results.

There are tools based on simulation that need to know when the steady state of the simulation starts, i.e., the warm-up period must be specified by the user and will typically be based on observations of pilots of the model. An example of such tools is JINQS, where there is no built-in mechanism for detecting when a simulation is close to a steady state. In this tool, when the approximate warm-up period is known, the simulation method can be optionally parameterized by this warm-up period.

Other tools implement some sophisticated methods to detect when the steady state is reached. In this way, JMT implements transient detection using the R5 heuristic [Fis73] and the MSER-5 [WCS00] stationarity rule. Then it uses variable batch sizes and a fixed amount of memory until the confidence intervals are generated with enough accuracy. It can also perform long-run simulations for the case of models with heavily-tail distributions. Some other tools also use the batch means methods, like TANGRAM-II [dSeSL00], or WinPEPSY-QNS [BG09], and so does ours.

Some tools offer different ways to determine the accuracy and confidence level of the solution. For example, JINQS offers an optional parameter called confidence interval. If none is supplied, a value of 0.05 is assumed. If the logged measures are dependent and/or are not normally distributed, the computed

Table 5.2: Simulation features of some packages and tools for QN modeling and analysis

Tool	Results presentation	Stopping criteria	Accuracy / Confidence interval
RESQME	Graphical and tabular results, and animation of the original diagram	Offers several methods to determine simulation run lengths: simulated time, number of departures from specified queues or nodes, etc.	User-defined confidence interval
SHARPE	Collection of visualization routines to analyze output results; results can be plotted. Excel spreadsheets can also be generated.	Steady state and transient computations	User-defined precision level (number of digits).
QNAP2	Screen textual output. Results can be saved in files	Users can simulate until some confidence interval is reached or a given simulation time expires	User-defined confidence interval
QSIM	Graphical interface	Simulation length control	Up to 95% confidence interval
SPE·ED	Graphical-interface	Batch means, simulation length control, number of jobs	User-defined confidence interval
PEPSY-QNS	Textual files	Offers several methods for identifying steady states, including *batch means*	Some methods require users to input the desired accuracy
TANGRAM-II	Textual files generated during the simulation	Offers several methods for identifying steady states, including *batch means*	User-defined confidence interval
PDQ	Results shown textually, by means C code (displayed in console or saved into a file	N/A (it only performs analysis)	Up to six decimal digits of precision
MQNA	Results are shown textually	Product-form QNs are solved analytically, and non product-form QNs are solved outside MQNA (PEPS and SMART use iterative methods)	Up to 10 digits precision in non-congested models
WinPEPSY-QNS	Textual files	Batch means	Errors smaller than 0.15% compared to the exact solution in several experiments
JINQS	Textual files	Warm-up period specified by the user; there is no built-in mechanism for detecting an approximate steady state	User-defined confidence interval
JMT	Graphical interface. Results can be exported in XML format	Implements transient detection using the R5 heuristic [Fis73] and the MSER-5 [WCS00] stationarity rule. Then it uses variable batch sizes. It can also perform long-run simulations for the case of models with heavy-tail distributions	User-defined confidence interval
qnetworks	Results returned as GNU Octave vectors or matrices, with values shown programmatically	Steady state and transient computations	Exact results for product-form QNs
xQNM	Textual files (*cvs* format)	Batch means or user-defined number of jobs	User defined confidence level

confidence interval will be inaccurate. If the replications are independent, mean values will be approximately normal, but variances and other measures may not be.

Normally, tools that carry out analytical methods offer a great accuracy. This is the case of QNAP2 [VP04], which satisfies the confidence interval introduced by the user; the PDQ Analyzer [Gun09, Gun05], which offers up to six decimal digits of precision; SHARPE [HSZT00, Tri02, ST87], whose output precision is determined by the option "Number of digits printed" in the output; or MQNA [BFS03], whose outputs have up to ten digits precision in non-congested models. As explained in the previous section, our tool uses the batch means method in each replication. Simulations start with a very small batch size and then they increment it until the variance of the batch means goes below 10^{-6}.

Results are presented in very different ways depending on the tool. Many tools display the results in plain text following some template, like QNAP2, TANGRAM-II, JINQS, PDQ and MQNA. Others are provided with a graphical user interface, which shows the results (SHARPE, JMT and WinPEPSY-QNS). RESQME [CGLM93, MG94, AGK+89] is even capable of animating the models as the discrete event simulations progress. Our tool outputs the results in a textual format readable by spreadsheet applications (*csv* format, Figure 5.14).

Regarding the time that these packages and tools take to get the performance properties, analytical methods are of course much faster than simulations. For example, QNAP2, PDQ, SHARPE, qnetworks or MQNA take less than a few seconds to obtain the results. On the contrary, RESQME, PEPSY-QNS and WinPEPSY-QNS may take from some minutes up to several hours to obtain the results, depending on the complexity of the input model. Our tool also uses simulation, and thus it may take from a few seconds to several hours depending on the size of the model.

5.4.3 Analysis comparison among tools

Once we have shown simulation and analysis features of some tools, in this section we run our case study in some of them to see the differences between them and our tool. The queuing network model is that of Figure 5.1(a). In this analysis comparison, we are going to focus on the performance measures obtained for DISK B.

For the analysis comparison, we have used WEASEL [SEA12] and JMT [BCS09]. WEASEL is based on PMIF and, consequently, the available elements to be drawn and the configuration parameters for those elements are very similar to those in xQNM. On the other hand, JMT is much more powerful in terms of available elements and configuration parameters; it offers the possibility to include in the model many elements not specified in PMIF: forks, joins, delays, routing stations, etc. The configuration parameters for the elements are also much larger: different load strategies for servers, many routing strategies available, etc. Since we are only dealing with the PMIF capabilities, we only use a small subset

Table 5.3: Analysis comparison. RP: Routing Probabilities, NV: Number of Visits

Tool	Utilization	Throughput	Wait T.	Serv T.	Res T.	Queue L.
PDQ (RP)	0.09	03.0	0.003	0.03	0.0330	0.099
PDQ (NV)	0.72	24.0	0.056	0.03	0.0857	2.571
PEPSY (RP)	0.72	24.0	0.077	0.03	0.107	1.851
PEPSY (NV)	0.72	24.0	0.077	0.03	0.107	1.851
JMT (RP)	0.69	23.4	0.158	0.03	0.187	2.597
xQNM (RP)	0.72	23.8	0.076	0.03	0.106	2.869
Theoretical	0.72	24.0	0.077	0.03	0.107	2.571

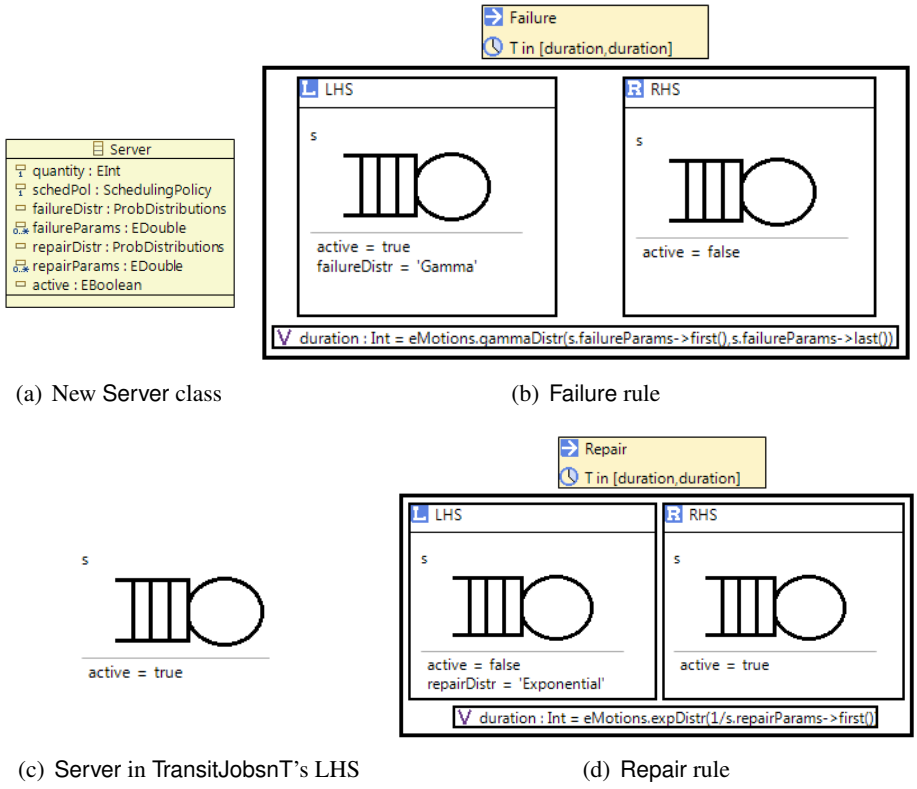
of JMT. The results for each tool run are shown in Table 5.3. The references used to compare the results of the different runs to check their accuracy were the theoretical results available in Jain's book [Jai91].

In the table, *RP* stands for *routing probabilities* and *NV* for *number of visits*. These correspond with the routing criteria followed in the runs. The PDQ Analyzer, executed by means of WEASEL, does not accept routing probabilities. Thus, when we run the experiment with routing probabilities (because it is possible to define routing probabilities with the graphical user interface of WEASEL, independently of the tool used afterwards to solve the models), the results were erroneous. This is because it only considers number of visits, so it considered that the number of visits in every server was 1. We then changed the criteria to number of visits, and the results were all the same as the reference apart from the waiting and residence times, which significantly differed. WEASEL offers the possibility to solve the models with PDQ using exact solutions, approximate solutions and canonical solutions. Our experiment was run with the canonical solution because the other two do not accept open networks.

The results with PEPSY-QNS [BK94] were also obtained by means of WEASEL. PEPSY-QNS offers different solving methods in WEASEL, and we used sopenpfn. We run the experiment with both number of visits and routing probabilities and they were the same, so this tool is capable of dealing with both. All the measures obtained with this tool coincided with the reference except for the queue length, which was smaller.

In WEASEL, it is not possible to specify the performance properties to be monitored. JMT accepts probabilities as routing strategy, apart from random, round robin, join the shortest queue, shortest R time, least utilization and fastest service. The output presented by the JMT tool is very intuitive, complete and easily readable. It offers statistics, including a chart, for each metric of each server. Furthermore, the user can specify which metrics he/she wants to monitor for which server.

The penultimate row of the table contains the results obtained with xQNM. The accuracy of the results is very good, which shows that the behavioral model that defines for QNM is faithful and accurate.



(a) New Server class (b) Failure rule

(c) Server in TransitJobsnT's LHS (d) Repair rule

Figure 5.15: Extensions for considering failures.

5.4.4 Considering failures

Once we have modeled the behavior of QNMs and are able to analyze their performance properties, we are interested in extending their behavior in order to consider more realistic situations. In this regard, we want to take into account failures and repairs in networks' servers, as they happen in real life. Thus, after starting all the servers operative, they can fail at some point and be inactive for a while before they are repaired and back to service. We have to consider times to failure and times to repair. These are normally modeled with exponential distributions [KK94], so that analytical calculations are possible. However, since we can include many probabilistic distributions to model this behavior, the modeler can choose any of them.

In order to extend the behavior of our DSL for modeling and analyzing QNMs with failures, we simply need to do two things: extend the ePMIF metamodel and add a couple of very simple behavioral rules. Only the Server class needs to be extended in order to include rates for failures and repairs in servers (Fig. 5.15(a)). The new attribute active is true whenever the server is operating, and false when it is not. Attributes failureDistr and repairDistr dictate the distribution followed by

the time failures and repairs happen, respectively, while attributes failureParams and repairParams contain the parameters of such distributions.

Rule Failure (Fig. 5.15(b)) models the failure of a server. It is the only rule that needs to be added for modeling such failures. In this case, the distribution followed by the time to failure is Gamma. It can follow any distribution in the ProbDistributions enumeration type (Fig. 5.4). In the rule's RHS the active attribute is turned to false, modeling the inactivity of the server. A slight modification needs to be carried out in the LHS of rule TransitJobsnT (Fig. 5.11) to launch it only if the server s is active (Fig. 5.15(c)). A similar rule is included for repairing a server, which takes a server which is inactive and activates it. Such rule is shown in Fig. 5.15(d), where it considers a repair rate that follows an Exponential distribution.

We have included these modifications and have carried out some experiments. We have made the times to failure and repair follow exponential distributions with rates 10 and 5, respectively. In general, jobs take longer in being processed and leaving the system, since they may need to wait in queues whose server is inactive, and have to wait until it is repaired. Furthermore, for the same arrival and services times of our case study [Jai91, page 572], the network does not satisfy Little's Law anymore, so analytical calculation becomes very hard and complex. The reason is that this example was created so that Little's Law was satisfied for the arrival and service times established, and considering that servers never fail. This is, the number of incoming and outgoing jobs per time unit (throughput) with the servers being active all the time was the same, 3, once the steady state was reached. However, since Little's Law is no longer satisfied in this example when server failures are taken into account, no steady state is reached, and the performance measures for our network depend now on the number of incoming jobs. Simulation becomes crucial in this case. Thus, we have carried out an experiment where 100 jobs enter (and leave after being processed) the network, and have checked that the performance measures significantly change, even for such a small number of jobs. The throughput value is now 2.22, and it will decrease as the number of incoming jobs increases due to contention in queues. The theoretical response time for the network without failures is 1.41, while the new response time considering failures is 2.33.

Although in the example we have considered the same failure and repair rates for every server, each one could have been modeled to have different rates, since every server can have its own characteristics (as it happens in reality). Similarly, different probabilistic distributions can be used and more realistic values for failure and repair times could also be set. This flexibility is one of the benefits that can be obtained by the use of appropriate DSLs for modeling complex systems.

5.5 Summary

In this chapter we have surveyed several tools for analyzing QNMs. We have built a semantic bridge by showing how QNMs can be interpreted in another modeling domain, in this case the one provided by *e-Motions* for specifying and simulating real-time systems. Having a representation of QN models in that domain has allowed the easy definition of a DSL for the specification and simulation of general QNMs, and the use of the tools available in that domain. In particular, our proposal has provided several interesting advantages and results.

First, a generic behavioral model for QNMs has been defined by means of six *e-Motions* rules. They provide a behavioral semantics for QNMs, expressed in a high-level language with precise semantics and execution facilities. Such behavioral model has been easily extended with two more rules in order to model failures and repairs in servers, which allows to analyze more realistic situations. This also shows how simple and flexible the behavioral model of the QNM can be changed when it is defined by means of a DSL, incorporating new features by simply adjusting some high-level rules.

Second, we have obtained a prototype tool that allows to draw QNMs, automatically translate them to their behavioral representation and finally simulate them. Models can be depicted graphically in xQNM, and they can be exported to PMIF 2 and ePMIF models. PMIF 2 models can also be imported to our tool in order to simulate them or to represent them graphically. The tool, together with a set of examples, is available from [TV12]. The use of MDE techniques has enabled a modular architecture, which can be easily maintained and extended in future versions, since each of its parts can be independently improved. We have also shown how the existing de-facto standard metamodel for QNM representation and interchange, PMIF, can be incorporated into the MDE domain, and easily extended to take into consideration more powerful and flexible possibilities and system properties.

6

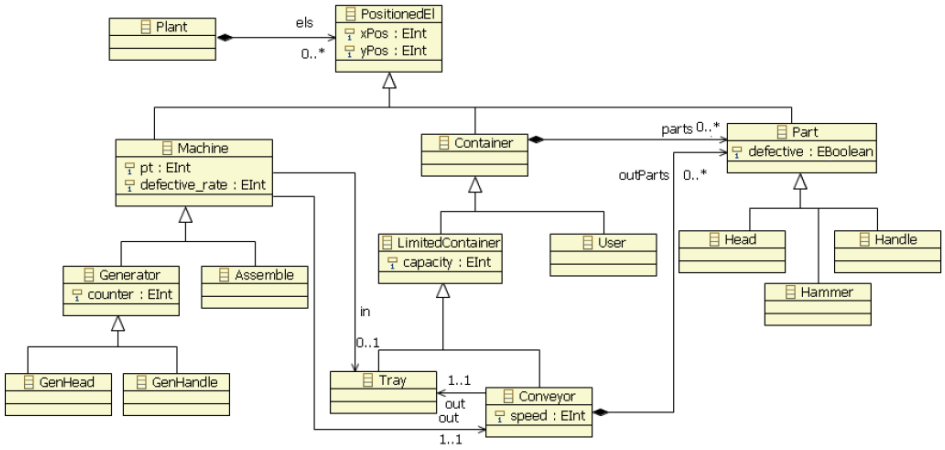
A Modular Approach for the Specification of Observers

As we have seen in previous chapters, observers allow a flexible way to monitor the system and to obtain information about its behavior. However, the addition of observers may require to change the existing behavioral models to a large extent, making system models potentially complex and difficult to maintain. One way to cope with this problem is by using aspect-oriented techniques [CB05], whereby a modular specification of the observers is provided, and then *woven* (in the aspect-oriented sense) into a concrete system.

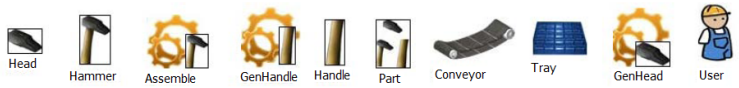
The goal is to define a library with different observer languages, which can then be reused by concrete systems to incorporate observers within their structural and behavioral specifications. This provides a modular approach to the specification and monitoring of QoS properties, for which observers can be independently defined and reused across system specifications. The definition of an observer language involves defining its abstract syntax (a metamodel), its concrete syntax (by assigning a visual icon to the observer) and its semantics (by means of one or more behavioral rules).

Our approach is based on standard software measurement approaches, which define *base* and *derived* measures [GBC+06]. The former ones allow measuring individual object attributes, while the latter build on the values of base measures to define aggregated metrics. Similarly, we propose *individual* and *general* observers. As mentioned in Section 3.2, individual observers are attached to individual objects to monitor their state and/or behavior. General observers, in turn, monitor individual observers, as well as the rest of the objects in the system, to build derived measures for non-functional properties such as throughput or mean time between failures, for instance.

Although of different nature, the behavior of individual and general observers share a similar and regular pattern, so they can be defined, independently of any system, using a similar approach. Then, individual observers are directly woven to concrete systems, incorporating the former in the latter. However, since general observers may monitor individual observers, it may be necessary to first weave general observers with individual ones and then weave the result to concrete systems.



(a) Abstract syntax (metamodel)



(b) Concrete syntax

Figure 6.1: Static structure for production line systems

The remainder of this chapter is structured as follows. Section 6.1 presents a motivating example for defining individual observers in a modular way. Section 6.2 explains how our approach is implemented, and Section 6.3 applies it to a different example. Section 6.4 describes and presents an example on how general observers are defined and woven to concrete systems. Finally, Section 8.1 summarizes the chapter.

6.1 Motivating Example

In this section, we present an example of what we want to achieve. It is based in the case study presented in Chapter 3. As it was explained, the definition of a DSL is made up of a metamodel of the language concepts, a concrete syntax and a set of transformation rules to specify the behavioral semantics of the DSL.

Figure 6.1(a) shows the metamodel already presented in Section 3.1. It allows to specify production line systems for producing hammers out of hammer heads and handles, which are generated in respective machines, and transported along the production line via conveyors and trays. Such metamodel provides the language's abstract syntax. In addition, a concrete syntax is provided. It is defined by providing icons for each concept (see Figure 6.1(b)).

Instances of this DSL are intended as token models [Tho06]. That is, they describe a specific situation and not the set of all possible situations (as is the case, e.g., for class diagrams). The behavioral semantics of the DSL can, as it was

seen in Chapter 3, be given by specifying how models can evolve; this is, what changes can occur in a particular situation. This is specified by a set of model transformation rules, with the form of those presented in Sections 2.2.2 and 3.1. Let us focus here in the production line system case study, and concretely in the Assemble rule (Fig. 6.2). It models how a new hammer is assembled: a hammer generator a has an incoming tray of parts and is connected to an outgoing conveyor belt. Whenever there is a handle and a head available, and there is space in the conveyor for at least one part (specified by an OCL constraint in the left-hand side of the rule), the hammer generator can assemble them into a hammer. The new hammer is added to the parts set of the outgoing conveyor belt. The complete semantics of this case study is constructed from a number of such rules covering all kinds of atomic steps than can occur.

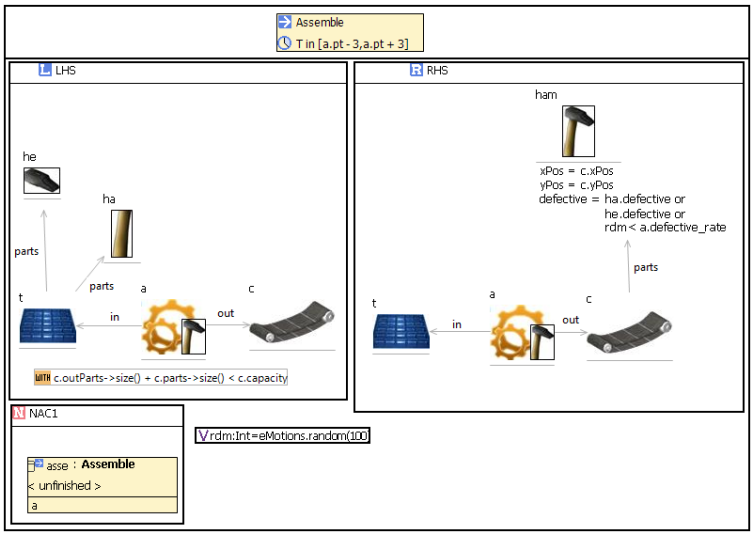
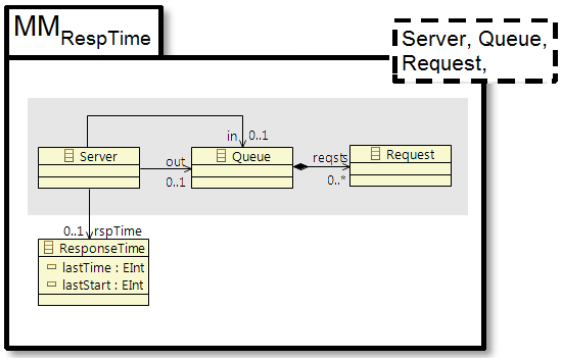


Figure 6.2: Assemble rule indicating how a new hammer is assembled

We are interested in monitoring certain non-functional properties of production line systems, such us the throughput or how long it takes for a hammer to be produced. We described in Chapter 3 an approach for extending the DSL specification with observers, which would be dealing with the monitoring of non-functional properties. In this chapter we suggest defining specification languages for observers entirely separately from any specific DSL. We will use the same mechanisms we used for defining the production line DSL to define a DSL that enables us to specify throughput or production time of systems.

Figure 6.3(a) shows the metamodel for a DSL for specifying production time. Two things should be noted about this metamodel:

1. It defines no concept of production time. Instead, it defines something called *response time*, which is a more generic concept. Production time is really



(a) Abstract syntax (metamodel) (b) Concrete syntax

Figure 6.3: Metamodel and concrete syntax for response time observers

only meaningful in the context of production systems. However, the general concept of response time covers this sufficiently well.

2. It is a parametric model (i.e., a model template). The concepts of Server, Queue, and Request and their interconnections are parameters of the metamodel, and they are shaded in grey for illustration purposes. We use them to describe in which situations response time can be specified, but these concepts will need to be mapped to concrete concepts in a specific DSL.

Figure 6.3(b) shows the concrete syntax for the response time observer object. Whenever that observer appears in a behavioral rule, it will be represented by that graphical symbol.

Figure 6.4 shows an example transformation rule defining the semantics of the response time observer. This states that if there is a server with an in queue and an out queue and there are initially some requests (at least one) in the in queue, and the out queue contains some requests after rule execution, the last response time should be recorded to have been equal to the time it took the rule to execute. Similar rules need to be written to capture other situations in which response time needs to be measured, for example, where a request stays at a server for some time, or where a server does not have an explicit in or out queue.

Note that the rule in Figure 6.4 looks different from the rule shown in Figure 6.2. This is because the rule is actually a rule transformation, while Figure 6.2 is a transformation rule. The upper part of Figure 6.4 (shaded in grey for illustration purposes) is a pattern or query describing transformation rules that need to be extended to include response-time accounting. The lower part describes the extensions that are required. So, in addition to reading Figure 6.4 as a 'normal' transformation rule (as we have done in the previous paragraph), we can also read it as a rule transformation stating: "Find all rules that match the shaded pattern

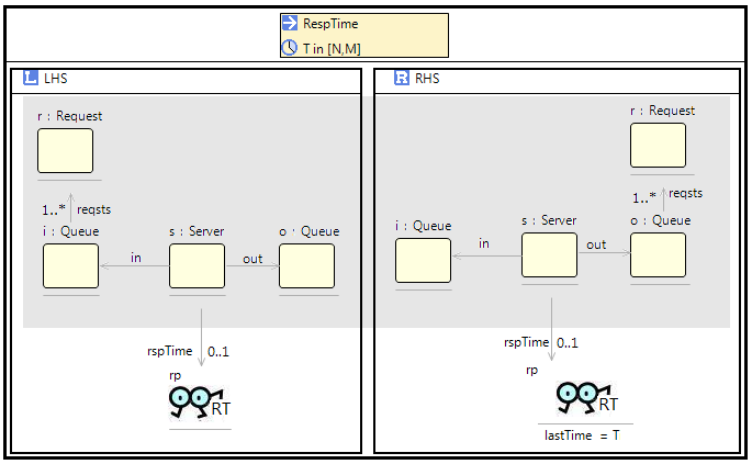


Figure 6.4: Sample RespTime rule

and add ResponseTime objects to their left and right-hand sides as described". In effect, observer models become higher-order transformations [TJF+09].

As the rules in observer models are rule transformations, we can allow some additional concepts to be expressed. For example, Figure 6.4 uses multiplicities to express that there may be an arbitrary number of requests (but at least one) associated with a queue. This is not allowed in 'normal' transformation rules (there we need to explicitly show each instance). However, using multiplicities allows expressing patterns to be matched against transformation rules—a match is given by any rule that has the indicated number of instances in its left- or right-hand side.

To use our response-time language to allow specification of production time of hammers in our production line DSL, we need to weave the two languages together. For this, we need to provide a binding from the parameters of the response-time metamodel (Figure 6.3(a)) to concepts in the production line metamodel (Figure 6.1(a)). Specifically, we bind:

- Server to Assemble, as we are interested in measuring response time of this particular machine.
- Queue to LimitedContainer, since the Assemble machine is to be connected to an arbitrary LimitedContainer for queuing incoming and outgoing parts.
- Request to Part, as Assemble only does something when there are Parts to be processed.
- Associations:
  - The in and out associations from Server to Queue are bound to the corresponding in and out associations from Machine to Tray and Conveyor, respectively.

– The association from Queue to Request is bound to the association from Container to Part.

Weaving the metamodels according to this binding produces the metamodel in Figure 6.5 – the added part has been shaded in grey. The weaving process has added the ResponseTime concept to the metamodel. Notice that the weaving process also ensures that only sensible woven metamodels can be produced: for a given binding of parameters, there needs to be a match between the constraints expressed in the observer metamodel and the DSL metamodel.

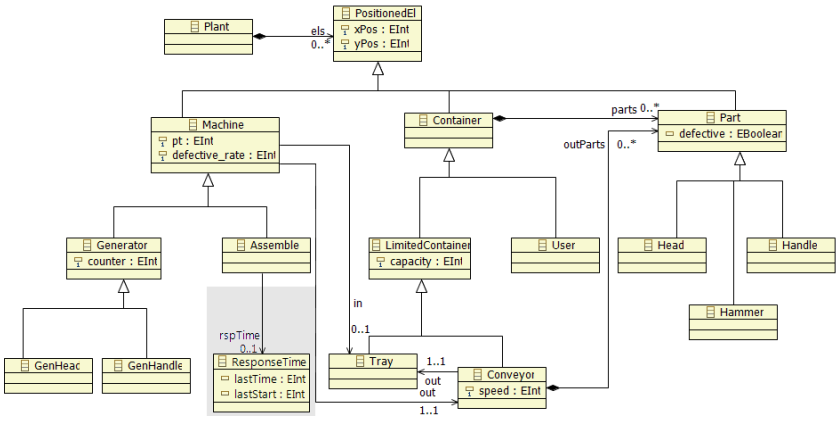


Figure 6.5: Metamodel result of the weave

The binding also enables us to execute the rule transformations specified in the observer language. For example, the rule in Figure 6.2 matches the pattern in Figure 6.4, given this binding: in the left-hand side, there is a Server (Assemble) with an in-Queue (Tray) that holds two Requests (Handle and Head) and an out-Queue (Conveyor). In the right-hand side, there is a Server (Assemble) with an in-Queue (Tray) and an out-Queue (Conveyor) that holds one Request (Hammer). Consequently, we can apply the rule transformation from Figure 6.4, which produces the rule shown in Figure 6.6 (the new objects and links have been shaded in grey). This rule is equivalent to what would have been written manually.

Clearly, such a separation of concerns between a specification of the base DSL and specifications of languages for non-functional properties is desirable. In the next section, we explain how we define correspondences between observer languages and specific systems and how we merge them together.

6.2 Implementation of our Approach

Figure 6.7 provides a graphical overview of the weaving process we are proposing. It can be seen that it consists of five parts (concrete syntaxes are not considered here for simplicity):

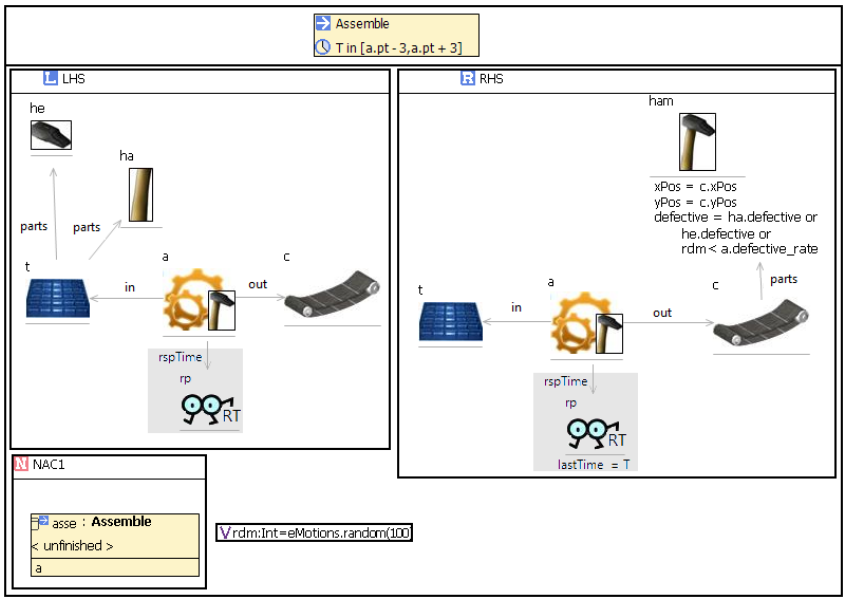


Figure 6.6: Result of weaving rules in Figure 6.2 and Figure 6.4

1. M_{DSL}: The specification of a DSL (without any notion of non-functional properties). It is composed of a metamodel (MM_{DSL}) and a set of behavioral rules (Rls_{DSL}).

2. M_{Obs}: The specification of a language for modeling non-functional properties of interest. It is composed of a metamodel (MM_{Obs}) and a set of behavioral rules (Rls_{Obs}).

3. *Binding*: An artefact expressing how the parameters of M_{Obs} should be instantiated with concepts from M_{DSL} in order to weave the two languages.

4. $\otimes$: A function that performs the actual weaving.

5. $M_{\widehat{DSL}}$: A DSL that combines the specification of some functionality (as per M_{DSL}) and some non-functional properties (as per M_{Obs}).

Following the proposal presented in Figure 6.7, we have split the binding process in two ATL transformations: one for weaving the metamodels, MM_{DSL} and MM_{Obs}, and another one for weaving the behavioral rules, Rls_{DSL} and Rls_{Obs}. In our case study, the former produces the metamodel shown in Figure 6.5, while the latter produces the woven rule depicted in Figure 6.6 (plus the remaining rules in Rls_{DSL}). For the remainder of this section, let us clarify that by *binding* we mean the relationships established between two models. As for *correspondence(s)* and *matching(s)*, we use them indistinctly when we refer to one or more specific relationships among the concepts in both models (between metamodels or behavioral rules). At the same time, we use concrete DSL and concrete system indistinctly.

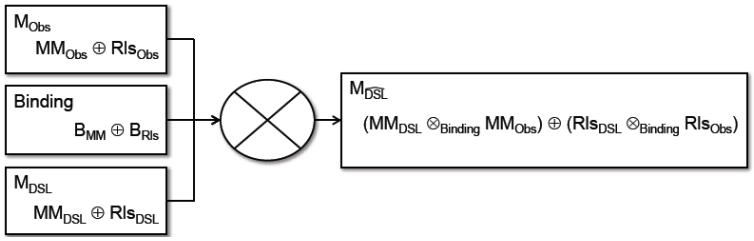


Figure 6.7: Architecture of the weaving process

The binding between M_{DSL} and M_{Obs} is given by a model that conforms to the correspondences metamodel shown in Figure 6.8. Thus, both bindings, between metamodels and between behavioral rules, are given in the same model. For the binding between metamodels (Figures 6.1(a) and 6.3(a) in our case study), we have the classes MMMatching, ClassMatching and RefMatching that specify it. We will have one object of type MMMatching for each pair of metamodels that we want to weave. In our example, we have one object of this type, and its attributes contain the names of the metamodels to weave. Objects of type MMMatching contain as many classes (objects of type ClassMatching) as correspondences between classes in both metamodels there are. Each object of type ClassMatching stores the names of the classes in both metamodels that correspond. We have three objects of this type, as described in Section 6.1. Regarding the objects of type RefMatching, contained in the refs reference from MMMatching, they store the matchings between references in both metamodels. Attributes obClassName and DSLClassName keep the names of the source classes, while obRefName and DSLRefName contain the names of the references. Once again, and as described in Section 6.1, there are three objects of this type in our example.

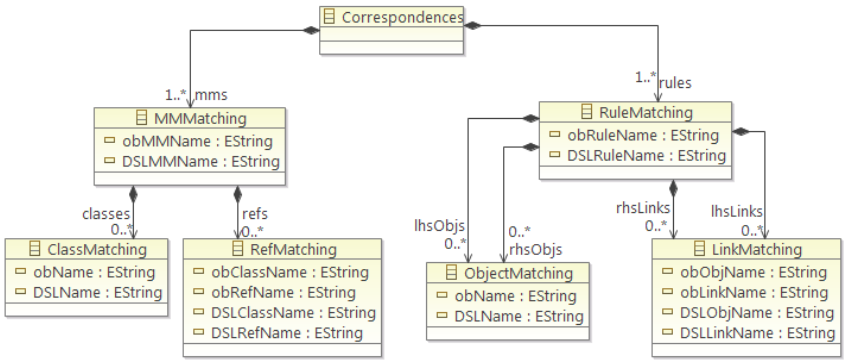


Figure 6.8: Correspondences metamodel

Regarding the binding between rules (Rls_{DSL} and Rls_{Obs}), there is an object of type RuleMatching for each pair of rules to weave, so in our example there is only one. It contains the names of both rules (Assemble and RespTime in

our case study). Objects of types ObjectMatching and LinkMatching contain the correspondences between objects and links, respectively, in the rules. Concretely, our correspondences metamodel differentiates between the bindings established between left- and right-hand side in rules, as we describe later. In our behavioral rules described within *e-Motions*, which conform to the Behavior metamodel (presented in [RDV09] and Appendix C), the objects representing instances of classes are of type Object and they are identified by their id attribute, and the links between them are of type Link, identified by their name, input and output objects. Similar to the binding between metamodels, objects of type ObjectMatching contain the identifier of the objects matching, and instances of LinkMatching store information about matchings between links (they store the identifier of the source classes of the links as well as the name of the links). The correspondences between rules Assemble and RespTime are those described at the end of Section 6.1.

As shown in Figure 6.9, we have split the overall weaving function into two model transformations, one for weaving the metamodels and concrete syntaxes and the other one for weaving the rules. Apart from the models already presented in Figure 6.7, GCS models (graphical concrete syntax) also take part in the transformations. They store information about the concrete syntaxes of DSL and observer models. Both transformations work in two stages to ensure the original DSL semantics are preserved. First, they copy the original DSL model into the output model. Second, any additions from the observer model are performed according to the binding information from the correspondences model. In the following subsections, we give a slightly technical explanation of both transformations separately. Their implementation is shown in Appendix B.

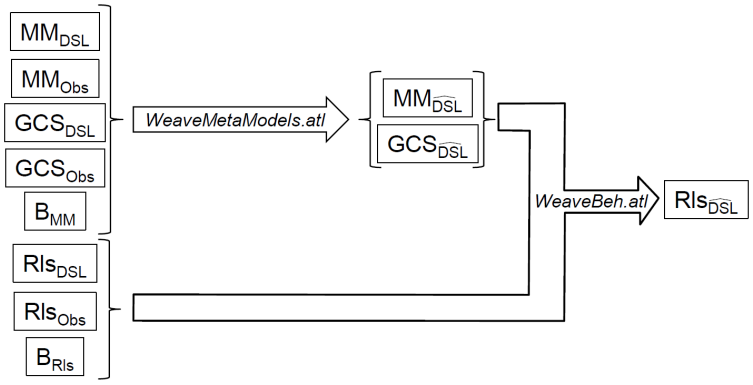


Figure 6.9: Transformations schema

6.2.1 Weaving metamodels

As mentioned before and as explained in detail in [Riv10], in *e-Motions* the concrete syntax of the concepts of a metamodel (GCS file) points to the file containing the metamodel (Ecore file). For this reason, in our case study we want to get

from this transformation, named *WeaveMetaModels.atl*, a file with the metamodel with the weave of the production line and response time metamodels ($MM_{\widehat{DSL}}$ in Fig. 6.9), which conforms to Ecore, as well as a file with its concrete syntax ($GCS_{\widehat{DSL}}$), conforming to the GCS metamodel. These two files are the output of the *WeaveMetaModels.atl* transformation (Fig. 6.9). This is the reason why we need as input in our transformation the GCS files with the concrete syntaxes of both input metamodels. Hence, the inputs of this transformation are the files containing the concrete syntax of the DSL and observer (GCS_{DSL} and GCS_{Obs}, respectively); the DSL and observer metamodels (MM_{DSL} and MM_{Obs}, respectively), conforming to the Ecore metamodel; and the model with the correspondences (B_{MM}), conforming to the correspondences metamodel exaplained before (Fig. 6.8). The Ecore and GCS metamodels are shown in Appendix C.

Before going through each part of the transformation, let us set its context. In the process of weaving both metamodels, we want to merge together those classes and references of the metamodels whose correspondences are specified in the correspondences model. Since all the concepts present in the observer metamodel (MM_{Obs} in Fig. 6.9) represent parameters, except those describing the actual observers and their references, merging together both metamodels means that the concepts in the DSL metamodel (MM_{DSL} in Fig. 6.9) will "prevail" over those parametric concepts in MM_{Obs} (in fact, the latter are instantiated by the former). In our example, Assemble (Fig. 6.1(a)) will prevail over Server (Fig. 6.3(a)), LimitedContainer over Queue, Part over Request, and the same for their references. Taking this into account and regarding the output woven metamodel ($MM_{\widehat{DSL}}$), we could think of a transformation where the input DSL metamodel (MM_{DSL}) would act as well as output of the transformation, and we simply need to decorate such metamodel with those concepts in MM_{Obs} representing observers. However, the input models of an ATL transformation are only readable, and the output models are only writable. It means that we cannot edit input models. Having this in mind, we explain the *WeaveMetaModels.atl* transformation by describing in detail each of its three parts:

1. Rules for copying the DSL metamodel. This first block of the transformation deals with the copy of MM_{DSL} into $MM_{\widehat{DSL}}$. It is done in order to decorate the latter afterwards, concretely, in the third part of the transformation. We need to copy every concept in the Ecore metamodel: EClass, EAttribute, EReference, EDataType, EParameter, EOperation, etc.

2. Rules for copying both the DSL and observer GCS models ($GCS_{DSL}$and GCS_{Obs}). This block merges the inputs GCS_{DSL} and GCS_{Obs} into the output woven GCS model ($GCS_{\widehat{DSL}}$). This part is composed of three rules. The first of them, CopyPackagesGCS, creates the package that will contain the output GCS model. It acquires its properties, such as its name, from GCS_{DSL}, since this one prevails over GCS_{Obs}. The second rule, CopyClassesGDDSL, copies the DSL classGDs, which contain the information about the concrete

syntax of the metamodel concepts as well as the references to the actual classes in the metamodel, in $GCS_{\widehat{DSL}}$. The class attribute, which has to point to the class in the output metamodel ($MM_{\widehat{DSL}}$), is retrieved by the ATL engine by searching for it in the traceability links. The class it looks for should have been created before by the CopyClasses rule in the first block. The third and last rule, CopyClassesGDObs, copies the observer ClassGDs of those classes representing non-functional properties. To find those classes, it checks that the class name is not equal to any obName attribute in any ClassMatching class in the correspondences model. To retrieve the class in the output metamodel to which the created ClassGD has to point, we make use of the CreateNonFunctionalClass lazy rule, explained in the third block of the transformation.

3. Rules for creating references, classes and attributes of non-functional properties. This part of the transformation looks for those classes, references and attributes in MM_{Obs} which represent non-functional properties, and copies them in $MM_{\widehat{DSL}}$. The main rule, CreateObserverOnlyReference, creates, in $MM_{\widehat{DSL}}$, those EReferences that point to a non-functional class in MM_{Obs}. In order to do it, the from part of the rule looks for those references in MM_{Obs} that do not have a matching in the correspondences model (they are not referenced by any object of type RefMatching) and creates an EReference in $MM_{\widehat{DSL}}$ with the same properties except for two of them: the class to which the reference points, established by the eType attribute, and the source class of the reference. The former class is created by means of the CreateNonFunctionalClass unique lazy rule, since it can happen that two EReferences point to the same non-functional class in the observer metamodel, but the class has to be created only once in the output metamodel. To assign the source class of the reference, we cannot use the eContainingClass property of the EReference since it is non changeable, so we look in our global variable that contains all the classes created (thisModule.classes) for the appropriate one and assign the EReference created as one of its eStructuralFeatures (the reference between EReference and EClass with the endpoints eContainingClass and eStructuralFeatures in Ecore is bidirectional – see the Ecore metamodel in Appendix C). The unique lazy rule CreateNonFunctionalClass receives an EClass representing a non-functional property of MM_{Obs} and creates that class in $MM_{\widehat{DSL}}$. The first call to this rule will create the class, and the remaining calls will return the class created the first time (this is the purpose of unique lazy rules in ATL). In order to assign the EAttributes of the class created, we get the eStructuralFeatures of the input class. The ATL engine will look in the traceability links for those structural features created in the CreateNonFunctionalAttribute rule from those of the input class. This last rule looks for EAttributes whose containing classes do not have a correspondence (so that the class is a class representing a non-functional property)

and create the same attribute in $MM_{\widehat{DSL}}$. The containing classes for the created attributes will be those created by the CreateNonFunctionalClass unique lazy rule.

6.2.2 Weaving behavior

The inputs of the *WeaveBeh.atl* transformation are the DSL and observer sets of behavioral rules (Rls_{DSL} and Rls_{Obs} in Fig. 6.9, respectively), which conform to the Behavior metamodel (see Appendix C); the woven GCS model ($GCS_{\widehat{DSL}}$) resulting from the *WeaveMetaModels.atl* transformation, conforming to the GCS metamodel (Appendix C); the woven metamodel resulting from the *WeaveMetaModels.atl* transformation ($MM_{\widehat{DSL}}$), conforming to the Ecore metamodel (Appendix C); and the model with the correspondences (B_{Rls}), conforming to the correspondences metamodel shown in Figure 6.8. The output of the transformation is a model, $Rls_{\widehat{DSL}}$ in Fig. 6.9, which conforms to the Behavior metamodel, containing a set of behavioral rules. That set of rules contains all those DSL rules that do not have any correspondence with any observer rule, plus the woven rules resulting from the merge of DSL and observer rules as specified by the correspondences model. The Ecore, GCS and Behavior metamodels are shown in Appendix C.

The *WeaveBeh.atl* transformation is composed of three blocks that we explain below.

1. Rules for copying the DSL behavioral rules (Rls_{DSL}). As we did in the *WeaveMetaModels.atl* transformation, the first block here deals with the copy of all the DSL behavioral rules (and their contained elements) into the output behavioral model ($Rls_{\widehat{DSL}}$). This block also contains an entrypoint rule, where we establish the package that will contain the behavioral rules. Such package has a reference to the woven GCS model ($GCS_{\widehat{DSL}}$), what makes the objects in the rules created appear with their concrete syntax. Many elements are to be copied in this block (see Behavior metamodel in Appendix C), namely AtomicRule, OngoingRule, Helpers, Variables, Patterns, ActionExec, etc. When copying Objects, their reference to the class they belong, given by its attribute classGD, has to point to that class in the input woven GCS model ($GCS_{\widehat{DSL}}$). As for Links, they contain an attribute, ref, that has to point to the actual EReference in the input woven metamodel ($MM_{\widehat{DSL}}$).

2. Rules for including observer objects and links in those rules having correspondences. This part aims to complete those rules created in the first block that need the inclusion of observers. In our case study, it will decorate the rule in Figure 6.2 so that it becomes the rule in Figure 6.6. To do so, we need the information about the observer rule (Fig. 6.4) and the correspondences between this rule and the Assemble rule. Rules CreateObserversLHS and CreateObserversRHS in the transformation (see Appendix B) do basically

the same, but the former decorates the LHS of a rule while the latter decorates its RHS. This distinction eases the organization of the transformation and allows us to consider those cases where an observer only needs to be added in one of the two patterns: if the observer object is only in the LHS, it means that it will disappear after the rule execution; if it is only in the RHS, it means that it did not exist before but it will be added to the system after the execution of the rule. Whenever the CreateObserversLHS rule is fired, it inserts in the LHS of the rule created from the DSL rule (rDSL in the transformation) an observer object and a link going from an existing object in the rule to the observer created. Regarding the link created, most of its properties are copied from the link in the observer rule that goes from a parametric object, which has a correspondence with a concrete object in the DSL rule, to the observer. Regarding the ref attribute, which points to the actual EReference in the metamodel, we look for the appropriate EReference in our woven input metamodel ($MM_{\widehat{DSL}}$). As for the object created, its classGD has to point to the corresponding class in the input woven GCS model ($GCS_{\widehat{DSL}}$). Objects, especially in the RHS of a rule, may contain slots. To create slots in the observers created, we use the ATL collect() operation to create slots from those slots of the observer in the observer rule. Lazy rule CreateSlot deals with the creation of such slots.

3. Rules for including clocks in the woven rules. Since the majority of the observers require using the time elapsed in the system, we add the clock object in all those woven rules where it was not already included.

6.3 Application

In this section we apply the presented approach over a different concrete system. We are going to use the response time observer language defined in Section 6.1 in order to weave it with the check-in and boarding system that was presented in Section 2.2. Recall that such system models the process of check-in and boarding of passengers in an airport. It models how passengers arrive in the airport, are redirected to desks' queues, wait there, realize the check-in, and finally get on their plane after waiting in the plane's queue. We want to apply our approach to record how personnel behind check-in desk perform. To such end, we want to associate response time observers to them. In this case, the response time observer would represent the concept of serving time (instead of production time, as in the production line system example).

The metamodel of the system is the one presented in Figure 6.10. We want to weave it with the response time observer metamodel shown in Figure 6.3(a). The first step is to define the correspondences model, which conforms to the correspondences metamodel (Fig. 6.8). Concretely, let us first define the part where the correspondences between metamodels are defined. We create an object of type

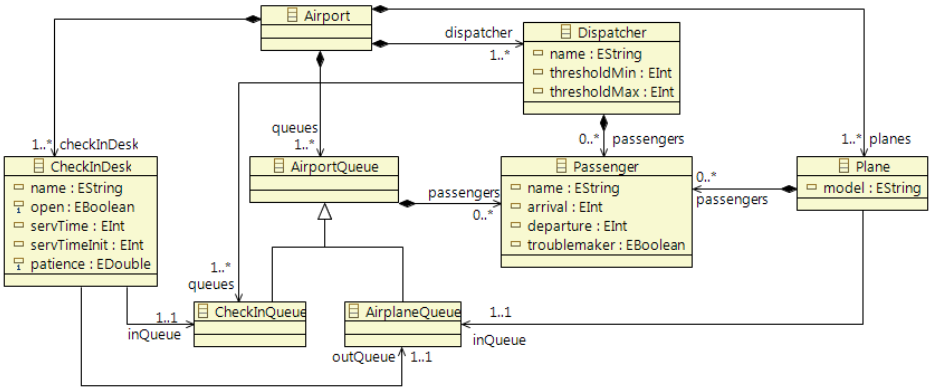


Figure 6.10: Airport metamodel

MMMatching and three of type ClassMatching. The latter are used to match Server with CheckInDesk, Queue with AirportQueue and Request with Passenger. We also need three objects of type RefMatching to specify the correspondences between references. We match the inQueue and outQueue references departing from CheckInDesk with the in and out references departing from Server, respectively. The passenger reference, departing from AirportQueue to Passenger, corresponds to the reqsts reference going from Queue to Request. We apply the *WeaveMetaModels.atl* transformation described in the previous section with both metamodels, the graphical syntaxes of both systems, and the correspondences model as input, and we get the woven metamodel displayed in Figure 6.11 (the class and reference for non-functional properties have been shaded in grey for illustration purposes) and the woven concrete syntax.

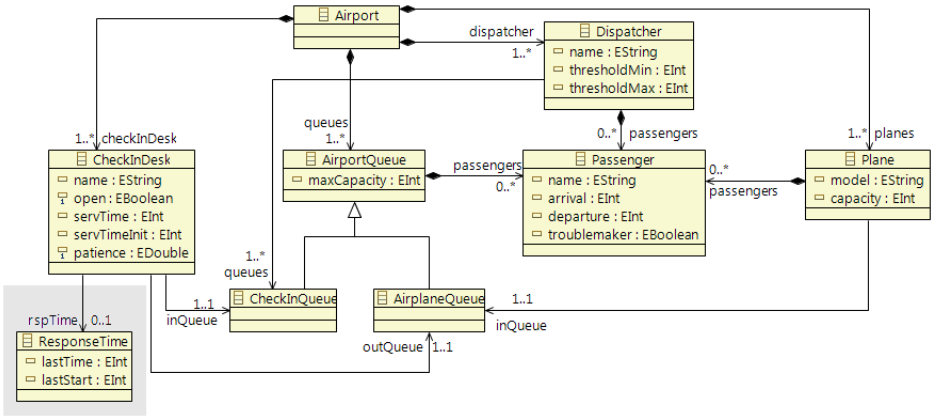


Figure 6.11: Woven metamodel for measuring response time of check-in desks

The next step is to include the response time observer in the rules where the check-in desks participate. In our example, we have the CheckInPassenger rule

(Fig. 6.12). It models how the first passenger in the desk's queue realizes the check-in and goes to the queue of his/her plane.

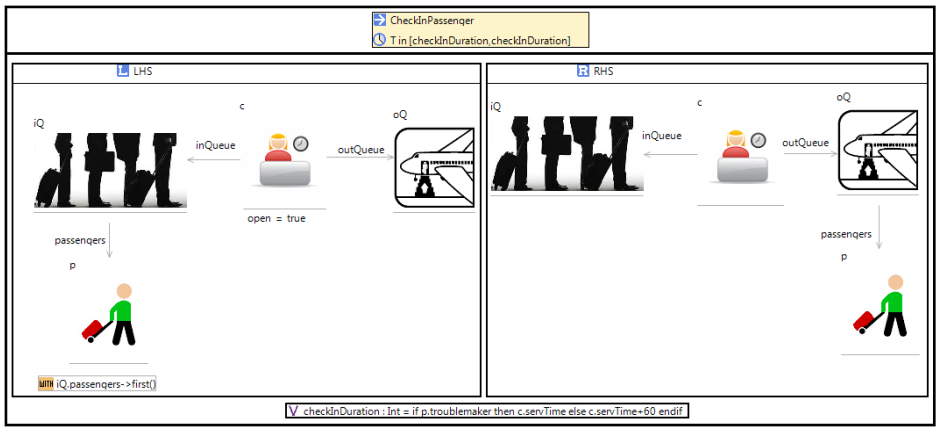


Figure 6.12: CheckInPassenger rule

In order to include the response time observer associated with the check-in desk in the rule, we first have to define the correspondences between this rule and the RespTime rule (Fig. 6.4). In our correspondences model, we add an object of type RuleMatching and four of type ObjectMatching for each pattern (LHS and RHS). The latter define the correspondences between the objects within the rules. In this way, r corresponds with p, i with iQ, s with c and o with oQ. We also need three objects of type LinkMatching for each pattern to define the correspondences between links in rules. We then apply the *WeaveBeh.atl* transformation described in the previous section with the woven metamodel and concrete syntax obtained with the *WeaveMetaModels.atl* transformation, the observer and system rules, and the correspondences model as input; and we get a set of behavioral rules, where the rule shown in Figure 6.13 (the objects and links for non-functional properties have been shaded in grey), result of the weave, is included.

With this we have proved that the same generic observer language can be used to monitor non-functional properties in two different DSLs.

6.4 Weaving General Observers to Concrete Systems

In most cases and as we saw in Section 3.2.2, general observers are used to monitor individual ones and build derived measures for overall systems' non-functional properties such as throughput, mean time between failures or mean response time, for instance. In this section we present an example for adding a general observer to measure the mean production time of all the assemblers in a production line system. Our general observer, that is to measure the generic concept of *mean response time*, will be updated whenever any individual observer monitoring response time changes its values.

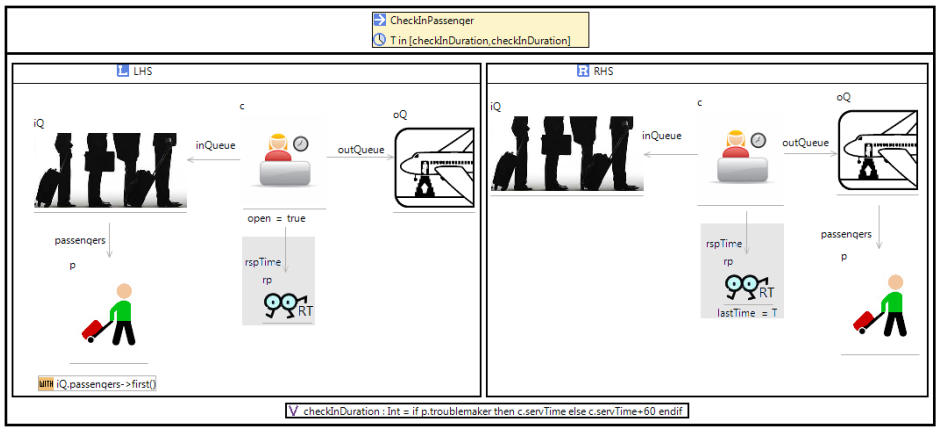
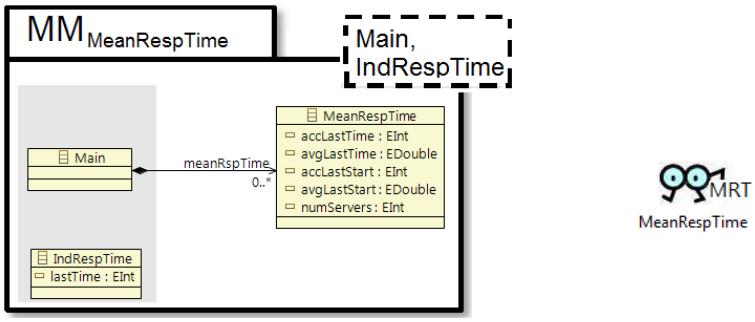


Figure 6.13: Woven CheckInPassenger rule

The first thing to do is to define the language for our new general observer: abstract and concrete syntaxes and semantics. The abstract syntax of general observers is quite simple because they are not to be associated to any class in a concrete system: they simply need to be contained by the main class of the concrete system (the class representing the system, which contains the remaining classes and which we never include in the behavioral rules because it represents the context). Abstract and concrete syntaxes are shown in Figure 6.14. Regarding the metamodel, it simply needs two classes apart from the one representing the observer. These classes, which are parameters (and shaded in grey in the figure for illustration purposes), are one named Main and one representing an individual observer. The former represents the class we have mentioned before, so it will be woven with the main class of a concrete system. As for the class representing an individual observer, it is the observer whose values will be used to do calculations in the general one. It includes an attribute, whose value is to be retrieved by the general observer. The name of the attribute could be any and changed in the weaving process, but we have chosen lastTime to make things clearer.

The semantics for general observers is also very simple, since general observers are updated when some individual observers are. Hence, we define as semantics for our general observer the rule shown in Figure 6.15. The rule shows that the general observer updates its values in a rule where an individual observer participates. It updates two of its attributes, according to an attribute of the individual observer. Attribute accLastTime is an accumulator and avgLastTime contains an average value.

The next step is to weave our mean response time observer language with a concrete system, our production line system. Since we need the existence of an individual observer in order for our general observer to make sense, the weaving process contains now two steps. In the first step, we weave the languages of both observers, and in the second step we weave the resulting language with a concrete



(a) Abstract syntax (metamodel) (b) Concrete syntax

Figure 6.14: Metamodel and concrete syntax for a general *mean response time* observer

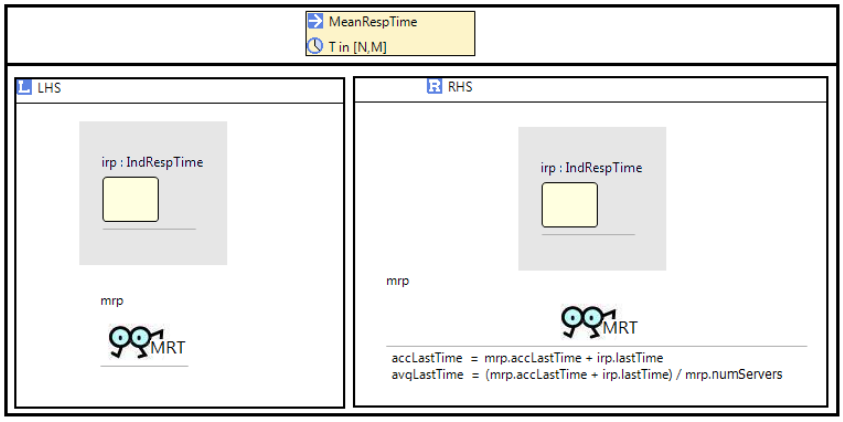
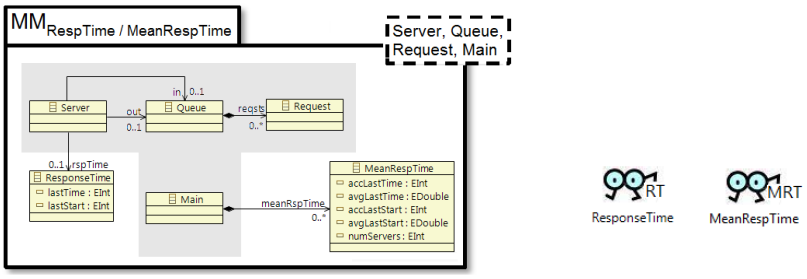


Figure 6.15: Sample MTRRule

system. Both steps are presented separately in the following subsections.

6.4.1 Weaving Observers Languages

Like it was done before, we first have to weave the abstract and concrete syntaxes of both observers (Figures 6.14 and 6.3). The metamodel for the general observer (Fig. 6.14(a)) is going to act as MM_{Obs} (see the transformation schema in Figure 6.9), since it is the one whose parametric classes are going to be instantiated with classes in the metamodel of Figure 6.3(a), so the latter acts as MM_{DSL}. In the correspondences model for weaving these metamodels, we simply have one object of type MMMatching, for relating both metamodels, and one of type ClassMatching, which relates the generic class IndRespTime in Fig. 6.14(a) with the concrete class ResponseTime in Fig. 6.3(a). The concrete and abstract syntaxes resulting from applying the *WeaveMetaModels.atl* transformation are shown in Figure 6.16.



(a) Abstract syntax (metamodel) (b) Concrete syntax

Figure 6.16: Metamodel and concrete syntax for a *general* and an *individual response time* observers

As we can see, those parametric classes in the metamodel acting as MM_{DSL} stay untouched, since they will be woven with specific classes of a concrete system in the next step. There is also a generic class in MM_{Obs} which was not matched to any concrete class in MM_{DSL}, Main, so that class also remains generic. It will also be woven with a specific class in the next step.

The next step is to apply the *WeaveBeh.atl* transformation to weave the rules of both languages: that in Figure 6.15 with the one in Figure 6.4. Once again, the former rule acts as Rls_{Obs} because we want to instantiate its generic individual observer with a concrete one: the one in Fig. 6.4, which acts as Rls_{DSL}. In the correspondences model, we create an object of type RuleMatching, and one of type ObjectMatching for each pattern. Regarding the latter, they relate irp with rp. The rule resulting from the weave is shown in Figure 6.17.

Once again, the parametric objects in Rls_{DSL} stay as such, since they are going to be instantiated with objects in a concrete system in the next step. Notice that the values of the slots of the mrp observer have slightly changed. In the parametric rule MeanRespTime, the update of attributes accLastTime and avgLastTime use the value of the lastTime attribute of the generic object ip. When the weaving is done, that value has to correspond with a value of the concrete object in the RHS of the woven rule, since the general observer requires the updated value of the individual one. If in a generic rule we want to get the value of an object in the LHS, then we use a variable and assign the value of the required attribute.

The concrete and abstract syntaxes in Figure 6.16 and the behavioral rule in Figure 6.17 conform a new language for an observer. In fact, it is a language which contains two observers, one for measuring the response time of each server and the other one for measuring the overall system response time. Since we want to measure the production time of assemblers, both for each assembler and the overall production time of all of them, we now have to weave the new language created with our production line system (PLS). It is done in the next subsection.

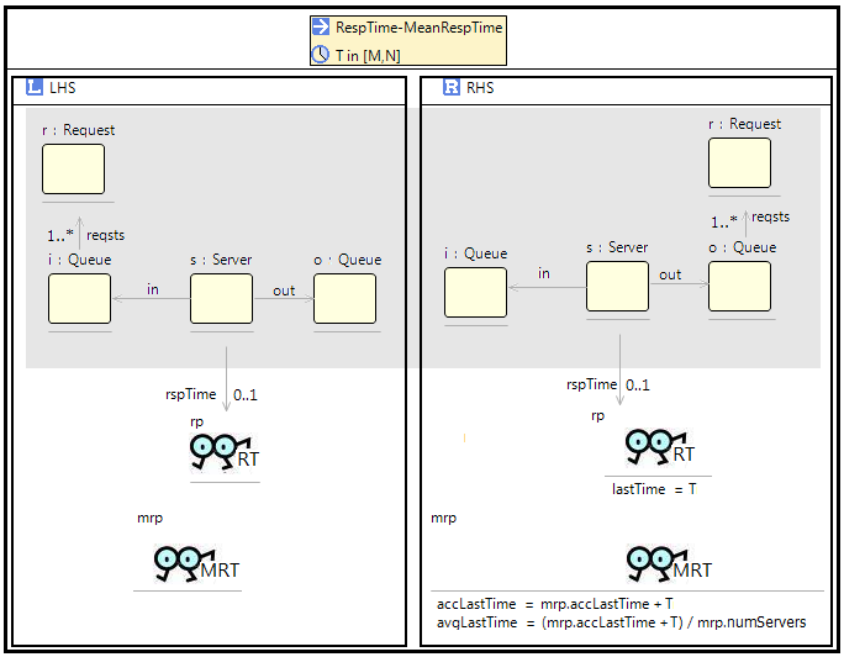


Figure 6.17: Result of weaving rules in Figures 6.15 and 6.4

6.4.2 Weaving the Resulting Observer Language with the PLS

Once we have the new language with both observers, we have to weave it with a concrete system. We weave it with our production-lime system in order to measure the production time of assemblers. The metamodel obtained in the previous step and shown in Figure 6.16(a), was the result of the *WeaveMetaModels.atl* transformation ($MM_{\widehat{DSL}}$). It acts now as input for the same transformation, with the role MM_{Obs}. It will be woven with the metamodel in Figure 6.1(a), which acts as MM_{DSL}. In the correspondences model, we match Server with Assemble, Queue with LimitedContainer, Request with Part and Main with Plant. There are also correspondences for all the references in the observers metamodel except the meanRespTime, since that one is not generic and is to be included in the woven metamodel. The resulting metamodel from applying the transformation is shown in Figure 6.18, and the added classes are shaded in grey for illustration purposes. We can see that the classes representing both metamodels, as well as their attributes and references, have been added all at once.

The last step is to weave rule RespTime-MeanRespTime (Figure 6.17) with rule Assemble (Fig. 6.2) in order to include observers in the latter. For that, the same correspondences as the ones explained in Section 6.1 between the RespTime and Assemble rules have to be defined. The RespTime-MeanRespTime rule is the output of the *WeaveBeh.atl* transformation in the previous subsection, so it acted as $Rls_{\widehat{DSL}}$ (Fig. 6.9). It is now acting as Rls_{Obs}, and the Assemble rule is

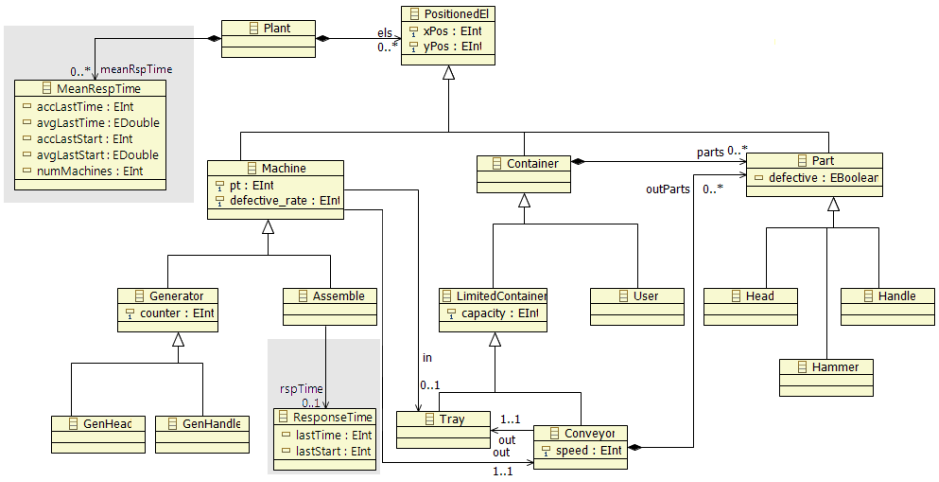


Figure 6.18: Woven metamodel including both observers

Rls_{DSL}. The result of the transformation is the rule shown in Figure 6.19, which contains the individual and general observers (shaded in grey for illustration purposes).

6.5 Summary

In this chapter we have presented an alternative approach to that in Section 3.2 for adding observers to a DSL. In this new approach we have proposed the independent specification of observers from any DSL. Hence, the idea is to have libraries where languages for different observers are defined. Said languages are then to be woven with concrete DSLs, so the latter are enriched with QoS monitoring capabilities.

Each observer language is a DSL itself, so they are defined by means of their abstract syntax (a metamodel), their concrete syntax (a visual icon for the observer) and their semantics (one or more behavioral rules). These languages contain parametric concepts, since they are later instantiated with concrete concepts when the weave with concrete DSLs is performed.

We have successfully applied our approach, implemented it by means of a correspondences metamodel and two ATL transformations (available in Appendix B), by weaving the same observer language, where an individual observer for monitoring *response time* of servers is defined, with two different concrete DSLs. In one of them, the individual observer monitors the concept of *production time* of assemblers in a hammers production line system. In the other one, we use the observer to monitor the *serving time* of personnel behind check-in desks in a system that models the process of check-in and boarding of passengers in airports.

Finally, we have also presented how this approach is also valid for weaving observer languages between them. This is specially useful when a general observer

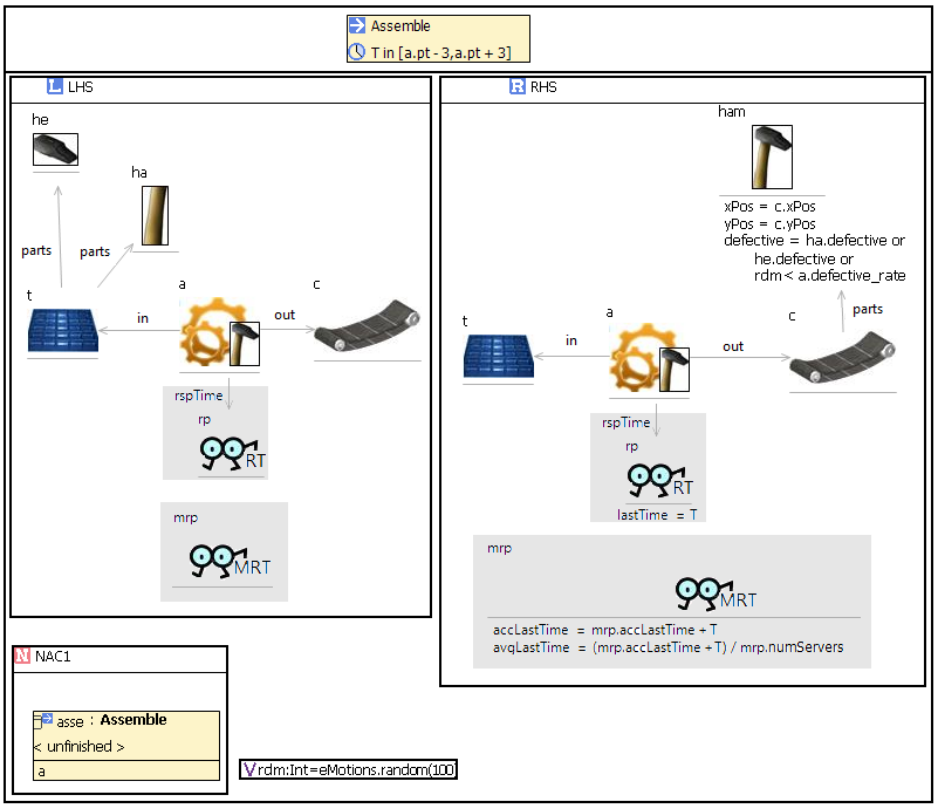


Figure 6.19: Woven Assemble rule including both observers

needs an individual one to update its state. We have shown an example when this case is useful, by weaving the language of a general *mean response time* observer with the language of an individual *response time* observer. The resulting language, which contains two observers and parametric concepts, has been successfully woven with the production line system, so that both observers are included in the DSL of the latter.

7 Related Work

The work presented in this dissertation focuses on the model-driven specification, modeling, obtaining and analysis of performance and reliability properties of dynamic systems in early phases of their development. The approach, as presented in the previous chapters, proposes the use of domain-specific (visual) languages to describe and specify both the functional and non-functional properties of systems. Although all the chapters are somehow correlated, each one of them makes a different contribution to the MDE and non-functional requirements analysis communities. For this reason, this chapter is structured in three sections. Section 7.1 compares our general approach for the high-level analysis and monitoring of non-functional properties with other works. We focus on wireless sensor networks in the last part of the section, since Chapter 4 presents our approach to model a WSN protocol and its reliability. Then, Section 7.2 describes works aimed at building the semantic gap between different semantic domains, since we presented a semantic bridge in Chapter 5 between low-level queuing network model specifications and *e-Motions*. Finally, Section 7.3 presents some related work in the general area of modular definition of languages, models and transformations.

7.1 Non-Functional Properties of High-Level Systems

The usual approach for specifying properties systems in order to analyze them through simulation consists of enriching the system elements with new states and several kinds of annotations. Although this might (partially) work for UML models, the situation is different when the models of the system are specified using domain specific visual languages, for which no clear solution currently exists. Most of the existing proposals for specifying QoS and other non-functional properties require skilled knowledge of specialized languages, which is precisely what DSVLs try to avoid with their notations closer to the end-users and to the problem domain.

There are analysis tools (such as ARENA [Roc11]) allow users to specify the models to be simulated using visual notations, but just within the tools environments and using their proprietary notations. In other words, these tools cannot easily take as input models produced by different editors, nor they can easily

export their models so that they can be analyzed by other tools. In addition, users cannot arbitrarily use these tools with their in-house developed visual languages, nor define in a flexible way the properties to be monitored and analyzed (let them be properties of the whole system or of any of its elements). In our proposal we separate the visual specification of the system from the tools that will be finally used to simulate or analyze them. Furthermore, the fact that we can use user-defined monitoring objects that are added to the system specifications for capturing the properties to be analyzed allows a high degree of flexibility, since it is the end-user who defines the observer objects as part of his/her system design.

Our observers were originally introduced in [TRV09] for specifying QoS properties, and in [TRV10] it was discussed how they could be used during simulations. In [TVDZ13] we showed how they can be effectively used for conducting performance analysis. We have also introduced the use of probability distributions, which gives us the possibility of analyzing stochastic systems, and the use of the batch means method for deciding when simulations become stable. Furthermore, a generic methodology that supports our approach has been presented. One of the benefits of our proposal is that it can be easily applied to other proposals that also advocate the use of in-place rules for specifying the behavior of real-time systems [GHV02, BÖ10, BGH$^+$05, dLV10], and specially those that propose visual notations for doing so.

Observers are not a new concept. They have been defined in different proposals for monitoring the execution of systems and to reason about some of their properties. In fact, the OMG defines different kinds of observers in the MARTE specification [OMG08]. Among them, TimedObservers are conceptual entities that define requirements and predictions for measures defined on an interval between a pair of user-defined observed events. They must be extended to define the measure that they collect (e.g., latency or jitter) and aim to provide a powerful mechanism to annotate and compare timing constraints over UML models against timing predictions provided by analysis tools. In this sense they are similar to our observers. The advantage of incorporating them into DSVLs using our approach is that we can also reason about their behavior and not only use them to describe requirements and constraints on models. In addition, we can use our observers to dynamically change the system behavior, in contrast with the more "static" nature of MARTE observers.

Regarding the model of time used, Boronat and Olveczky also present in [BÖ10] the use of in-place model transformations to complement metamodels with timed behavioral specifications. However, the way in which they model time is different. They add explicit constructs for defining time behavior and include time constructs in the system state whose semantics is encoded in MOMENT2. We, instead, do not add any explicit constructs for defining time behavior, but our transformation rules have time intervals denoting the duration interval of each local action. Moreover, any OCL expression and variable used within a rule can be used for denoting the duration of the rule, which permits us to

model stochastic systems if we use probability distributions for the durations. In [dLGB+10], de Lara et al. also use metamodels to describe the abstract syntax of systems. Then, they use DPO Graph Transformation rules [EEPT06] to specify its behavior. To introduce an implicit notion of simulation time, they extend their grammars with scheduling functions, associating edges with relative time values and with probability density functions. In that way, they can model either specific times as well as discrete and continuous distributions, such as the uniform, normal and exponential negative. They also propose how to measure some performance properties, although they do not propose a complete solution. We can use random values and any probability distribution in our rules, and not only for time durations but also for any other values within the rules.

Stochastic Graph Transformation systems were defined by Reiko Heckel in [HLM06]. They are an extension of graph transformations where the duration of the rules can follow probabilistic distributions. They have been used to study the performance and reliability of different kinds of systems [KH11]. At the beginning, this approach was based on the use of analytical methods, which generally requires some simplifications on the kinds of probability distributions they admit, and have also limitations to cope with the huge state spaces of current software systems. To overcome these limitations, more recent work in this field proposes the use of stochastic graph transformation simulations, which allows larger models and more general distributions. For example, in [HT10] Heckel and Torrini add a model of global time and apply stochastic simulations on mobile systems, and in [THR10] they use stochastic simulation in order to verify software performance in large-scale systems.

An alternative approach to relying on observers, as we do in our approach, is to rely on traces that can be generated from the transformations. The analysis of system execution traces to validate QoS properties has proved to be very effective in the case of component-based systems, network protocols and distributed middlewares [DBL+02, MMM02, SBHS06], and allows different kinds of very powerful analysis, as described in [HDSM05, Hil11]. But in fact, our observers can be used to generate selected sets of traces, as we have shown in Section 3.3.2, and then use these traces (stored in the observers' attributes) to analyze the system. The advantages in our approach is that the user, when defining the observers, can select the kind of execution traces he is interested in—instead of having the transformations generate all system traces, which can result in too much information.

As we do, Zschaler [Zsc10] proposes to consider non-functional properties in the development cycle as early as possible. He proposes a component-based approach which, as well as ours, models the system in a high-level manner. In [Zsc10], he shows a formal specification of timeliness properties of a component-based system, as an example for a formal approach to specifying non-functional properties. He explains how the different high-level parts of a component-based system work together to deliver a certain service with certain non-functional requirements.

Wireless Sensor Networks.

Regarding wireless sensor networks modeling, the work in [LVCA+07] presents by means of an MDE approach a DSVL to model WSNs, as we do. According to their approach, the WSN models described are then to be translated to WSNs domain specific textual languages by means of model-to-model and model-to-text transformations, in order to later simulate them. In this way, they only describe the static structure for WSNs, but not its dynamics. In fact, they add behavioral elements in the static models. In our approach we model the static and behavior analysis of the networks, and analyze their reliability. As far as we are concerned, our work is the first that proposes an approach to model and simulate WSNs using high-level DSVLs. Furthermore, no other work has previously presented the use of several RHSs based on probabilities in transformation rules, which allows us to model systems in a more realistic and precise way.

The works in [JN10, SWKS01, SS04] present and describe the directional source aware routing protocol (DSAP) and also compare some of its variants in terms of network lifetime. However, in none of these works it is mentioned the way the protocols are implemented, although we believe it is in a lower level than ours, nor the platform or program used to simulate them. They have not taken into account either real-life scenarios where power consumption in stand-by or death of nodes due to random circumstances are considered. Some other works have also studied the reliability in WSNs, such as the one presented in [AEH06], where they do consider the failure probability of each sensor. Their algorithm considers different types of topologies for the network, while the DSAP and variants only consider local information, and is implemented using dynamic programming to compute reliability. The RPAR protocol [CHX+06] is also a power-aware routing protocol for WSNs that tries to maximize the network lifetime by dynamically adapting packets forwarding.

As for what we have considered reliability in WSNs, there are similarities and differences when compared to other works. For example, the authors in [AEH06] define reliability of a WSN cluster as the probability that "a minimum aggregate rate of information can be delivered to the sink node". In many works [LCSW07, CDLR12, EPG+07], reliability in WSNs has to do with link reliability, this is, reliability between sensor nodes. Thus, probabilities of successfully sending/receiving packets for every pair of connected nodes have to be given. From such probabilities, the overall network reliability is calculated. In [CW10], the authors relate reliability to the percentage of data expected to arrive at the sink nodes. They also consider that nodes may run out of energy, be damaged or stolen, as we do. The authors in [TTL09] relate reliability to the success of every individual packet transmission. According to them, such reliability depends on the distance and the power dissipation in transmissions. Consequently, they try to minimize the power consumption, as our implemented protocol does. In other works, the authors speak about reliability in WSNs but do not give a concrete definition for it. As an example, the authors in [CHX+06] do not explicitly state

the way they define reliability, although the reader can deduce that it is related to the number of deadlines missed and energy consumption.

7.2 Bridging the Gap between Semantic Domains

Model-driven engineering changes the traditional view of software modeling, which no longer servers merely as documentation that eventually is put aside at a certain point during the development. Instead, MDE has made models an integral part of the development process and aims to bridge the semantic gap between the problem domain and solution domain. By bridging this gap, high-level representation of systems can be transformed into low-level implementations ready to be executed. In the construction of such bridges, model transformations play the most important role. We are interested in the gap bridging between DSLs and formal domains, so that the analysis capabilities of the latter can be used to analyze the former.

In [TV10, TV11] we propose a formal semantics for the ATL transformation language based on rewriting logics. To do so, we present a semantic mapping between the ATL and Maude languages. The purpose in the construction of such semantic bridge is to provide ATL transformations with formal analysis. This way, we can check the correctness of a transformation. Or, for example, after a transformation is executed, we can retrieve the rules which have not been fired, or we can identify the target elements obtained from certain source elements. Giese et al. [GHL10] also present the problem of model transformations lacking formal semantics in order for them to be correct and repeatable to support incremental development and maintenance of high quality software. They face the problem for the specific case of triple graph grammars (TGG) [SK08], which have a well understood formal semantics and are quite similar to other relational approaches such us QVT Relational [GK10]. For that, they bridge the gap between the formal semantics of triple graph grammars [Sch95] and the batch model transformation of the authors' related implementation [GH09]. With such implementation of a transformation, they can ensure that a valid rule set results in a unique and semantically correct outcome.

Zhang et al. [ZJBL08] propose an MDE-based approach to build bridges between informal, semi-formal and formal notations. To do so, they first consider different notations as different DSLs, so they introduce the notations into the ATLAS Model Management Architecture (AMMA) platform by the Kernel Metametamodel (KM3) [JB06] metamodeling. Then, ATL transformation rules are developed based on the built metamodels of the source and target notations. These rules define semantical mapping from the source metamodel to the target metamodel and are executed to obtain models consistent with the target metamodel. Finally, model-to-text rules are developed using TCS [JBK06], so that the generated models can be mapped to textual programs. The validity and practicability of the approach is shown by a case study of bridging OMG sys-

tems modeling language™ (OMG SysML) [OMG07] to Language Of Temporal Ordering Specification (LOTOS) [ISO89].

There are also approaches that propose to bridge the gap between internal data representation in different tools, at the same level of abstraction, like the work by Brunelière et al [BCC+10]. They present a model-based solution for interoperability in tools. The different tools are able to handle data (models) conforming to different metadata (metamodels). They propose solving syntactic and semantic differences between the tools. They do not try to jump to or from any formalism, but to automate the process for exchanging data between tools. In the paper, they present some bridges between Eclipse and Microsoft modeling tools, and plan to complete the bridges in future works.

In [dLV10], de Lara and Vangheluwe presented an interesting approach to automatically generate model-to-model transformations from DSVLs into semantic domains with a explicit notion of transition. From the initial model of the system and the behavioral rules, they generate a transformation which is expressed in the form of operational triple graph grammar rules. More precisely, they transform a production system described with a metamodel, an initial model and a set of behavioral rules into a Timed Petri Net (TPN). They allow the specification of the duration of rules as an interval between a lower and an upper bound. Such rule durations are then translated into durations in transitions in the TPNs. After translating a system to its representation in Petri nets, specific Petri net techniques can be used to analyze said system. However, in spite of the usefulness of this technique, it imposes some strict requirements on the behavioral rules in order to successfully transform them into Petri nets. Furthermore, it is not possible to use any probability distribution on the rules duration. We wanted to follow an approach similar to this for transforming our *e-Motions* specifications into queuing network models. However, we did not want to get rid of any feature in our behavioral rules, so they were too complex to be mapped to QNMs elements. For example, how could not find a way to map complex OCL expressions. This is why we decided to do the reverse mapping and make the most of the features gained in this semantic jump. In fact, after the jump, we can easily support arbitrary probabilistic distributions for arrival and service times, we can use (an Ecore version of) PMIF for defining and exchanging QN models between tools, or we can decorate the high-level QNMs specifications by for instance considering that servers can fail and be temporarily unavailable.

7.3 Modular Definition of Languages, Models and Transformations

In Chapter 6, we propose to weave two language definitions, one language enables the (abstract) specification of a set of non-functional properties while the second language focuses entirely on specifying relevant behaviors in a particular domain. Below we briefly review some related work in the general area of modular

definition of languages, models, and transformations.

7.3.1 Modular definition of languages

There is a large body of work on modularly defining computer languages. Most of this work (e.g., [BV04, BV09, KRV10]) deals with textual languages and in particular with issues of composing context-free grammars. While the general idea of language composition is relevant for our work, this specific strand of research is perhaps less related and will, therefore, not be discussed in more detail.

For languages based on metamodeling, there is much less research on language composition. Much of the work on model composition (see next sub-section) is of course of relevance as metamodels are models themselves. Christian Wende's work on role-based language composition [WTZ10] is an approach that specifically addresses the modularization of metamodels. For a language module, Wende's work allows the definition of a composition interface by allowing language designers to use two types of metamodel concepts: metaclasses and metaroles. The former are used as in normal metamodeling to express the core metamodel concepts. The latter are like metaclasses, however they actually represent concepts to be provided by another language—including definitions of operations and attributes, which are left abstract in the metarole. Metaroles are, thus, similar to our use of metamodel parameters in MM_{Obs}. However, Wende's work uses metaclass operations to provide an operational view on language semantics, while we use model transformations to encode language semantics.

7.3.2 Modular modeling

Our notation for expressing parametrized metamodels is based on how UML expresses parametrized models. Similar notations have been used in aspect-oriented modeling (AOM) approaches—for example, Theme/UML [CDJC08] or RAM [KAK09]. More generally, our language composition technique is based on the notion of model weaving from AOM. Theme/UML, RAM, or Reuseware [HHJZ09] are examples of aspect-oriented modeling techniques, which are asymmetric [HOT02]; that is, they make a distinction between a base model and an aspect model (the model that is parametrized) that is woven into the base model. This is also true of our approach: M_{DSL} is the base model and M_{Obs} is the model that is woven into it. There is an alternative approach to AOM that is more symmetric and considers all models to be woven as equal. This is typically based on identifying corresponding elements in different models and merging these. Examples are UML package merge or signature-based merging [RGF+06]. Most types of AOM also consider syntactic weaving only, disregarding the semantics of the modular models. In contrast, we explicitly consider the model semantics and provide formal notions ensuring that the composition does not restrict the set of behaviors modeled in the base DSL.

7.3.3 Modular model transformations

The semantics of the languages we are discussing are expressed using model transformations. As such, work on modularizing model transformations is of relevance to our work. Generally, this work can be distinguished into work on external and on internal modularization of model transformations. The former considers a complete model transformation as the unit of modularity, while the latter aims to provide modularity inside individual transformations [Kle06]. As we are modifying the internals of the base transformation by adding in detail concepts described in the observer transformation rules, our approach is an *internal* modularization technique. Nonetheless, ideas from external composition approaches are of interest to us. In particular, the work on model typing and reusable model transformations presented in [SMM+12] shows how the set of metamodel concepts effectively used by a model transformation can be computed and how this can be used to make the transformations more reusable. This is similar to the way in which we use the parametrized part of MM_{Obs} to make the observer transformation rules more reusable and to adapt them to different DSLs.

8 Conclusions and Future Work

This chapter is structured in two sections. Section 8.1 summarizes the advances presented throughout this dissertation in the field of high-level analysis and monitoring of non-functional properties of dynamic systems in early stages of their construction, as well as in the definition of semantic bridges. Then, we outline some ideas for future work in Section 8.2.

8.1 Summary and Contributions

We started our contributions in this dissertation by proposing an approach that tries to overcome a critical drawback found when constructing software applications or any kind of system. In the development of a system, whose life cycle was represented in Fig. 1.1, the changes that need to be done in the latter stages are frequently very expensive. Moreover, these changes are often propagated backwards, something which is not really desirable. Such changes need to be addressed because the system is not working as expected after it is deployed. The reason may be that its non-functional properties were not taken into account during the early stages of its development. In Chapters 3 and 4 we presented an approach for the light-weight modeling and testing of non-functional properties of dynamic systems in the design phase of the system construction. The approach is developed within the field and recommendations of the Model-Driven Engineering methodology, where domain-specific models (in our case, domain-specific visual models) are used as first class entities.

We propose the use of special objects, named *observers*, that can be added to the graphical specification of a system for describing and monitoring some of its non-functional properties. Observers allow extending the global state of the system with the variables that the designer wants to analyze when running the simulations, already in the design phase, being able to capture the non-functional properties of interest. Furthermore, the fact that action executions are first-class citizens in the *e-Motions* specifications can enable observers to monitor not only the state of the objects of the system but also their actions. In Chapter 2 we presented the *e-Motions* prototype tool for specifying systems using DSVLs. Such tool has served as proof-of-concept for our approach.

The idea is to model the non-functional properties of systems in the design phase, and be able to check such properties in that phase. This way, the design of the system can be redone until the system works as expected regarding its non-functional properties. In Chapter 3, we applied our approach in two case studies, a production line system and a network with packet switching. We showed how the behavior of a system throughout its whole life can be monitored, defining traces in observers, and how changing certain features of the system makes the simulations produce different results. The idea is to reconfigure different features of the system in order to obtain the desired non-functional results. In Chapter 3 we focused on performance properties, while in Chapter 4 we focused on reliability. In the latter we applied our approach in the domain of wireless sensor networks. We modeled a WSN protocol, named DSAP, and explained how different variants of the protocol can be modeled easily with our approach. Furthermore, we extended the behavior of the protocol by considering more realistic situations. Concretely, we considered energy consumption of nodes in stand-by and how nodes fail due to random circumstances. We showed how, taking into account these considerations, the reliability of the network is worse than in the original DSAP – which does not consider them. A remarkable contribution was made in the modeling of the system in Chapter 4 compared with the modeling in Chapter 3. It speeds up our simulations considerably. It consisted of modeling the objects that flow within the system as elements' attributes, instead of as elements themselves. Thus, hammer's parts (handles, heads and hammers) are modeled as elements in the production line case study, as well as the packets in the packet switching case study. However, packets are not modeled as elements in our WSN system, but they are modeled as attributes of the class node. This reduces considerably the number of objects in our models, so the matchings in Maude's rewriting rules are performed much faster and simulations conclude in shorter time.

After presenting our approach and applying it to different case studies, we used it to translate a formalism, queuing network models, into *e-Motions*. Effectively, we built a semantic bridge between the formalism and a DSVL represented in *e-Motions* and which uses our observers. Such DSVL consists of a generic set of behavioral rules that model different types of QNMs. These rules can be simulated to obtain the performance properties of the models. In this approach, the jobs, which flow among servers, are modeled as attributes, with the aim explained before. Furthermore, the rules can be decorated, so that the behavior of QNMs can be extended. In fact, we showed how the performance of a queuing network varies when considering failures in servers. A tool named xQNM has been created on top of the DSVL as an Eclipse plugin. The tool allows to draw QNMs conforming to a performance interchange format (PMIF), a metamodel created as a standard to exchange QNMs among tools. Actually, we adapted PMIF so that it conforms now to the Ecore metamodel and it can be integrated easily with Eclipse. Our version has been called ePMIF. After drawing a QNM, it can be simulated so that its performance properties are obtained. Many probability distributions can be used

for arrival and service times. The tool also allows to import models conforming to PMIF 2.0, and to export them in XML notation. This permits the exchange of models in xQNM with other tools that also support PMIF.

At this point, we think it is also worth mentioning a contribution of ours that we briefly described in Section 2.1.2 when presenting the ATL Transformation Language. It consists of the definition of a formal semantics for ATL based on rewriting logic. Such definition was implemented in Maude. The use of Maude as a target semantic domain brings very interesting benefits, because it enables the simulation of the ATL specifications and the formal analysis of the ATL programs. In particular, we showed how our specifications can make use of the Maude toolkit to reason about some properties of the ATL rules.

Finally, we presented an alternative approach to add observers into DSVLs. We consider it increases the modularity, flexibility and extendability in the definition of observers. It consists of defining DSVLs for observers independently from any system. These DSVLs are to be defined in a parametric way, where the parametric classes and references (in metamodels), and objects and links (in rules) are then instantiated by concrete concepts of a DSVL of a concrete system. Consequently, the independent definition of a DSVL for an observer consists of the definition of a parametric metamodel and a parametric set of behavioral rules (it may be composed of only one rule). Then, two bindings are performed. In the first one, the metamodel of the observer is mapped to the metamodel of a concrete system. This is, the parametric classes have to correspond with concrete classes in the system, and the same for the links. After this binding, the observers are included in the metamodel of the concrete system. In the second binding, the rules for the observers are mapped to rules of a concrete system. Thus, the parametric objects of the former are mapped to concrete objects in the latter, same thing for the references. After the binding, the observers are added in the behavioral rules of the system.

In Chapter 6 we presented this approach, and applied it for individual and general observers. We presented two case studies where the independent and parametric definition of an individual observer is woven afterwards with two different systems. Consequently, the same individual observer is automatically included in both systems, after defining the bindings with each of them. Since general observers often use aggregated values gathered from individual observers, we showed how it is possible, and actually convenient, to weave a DSVL of a general observer with a DSVL of an individual observer. In this case, the former acts as parametric DSVL, while the latter acts as a concrete DSVL. The result of this weave is a DSVL with the definition of two observers, where the general one uses values of the individual one to do calculations. Such DSVL is then woven with a concrete system, so that both observers are finally included in the system. Although the approach presented in the chapter has been actually implemented and proved in different case studies, it is still in a naive and early phase and many aspects should be studied and improved, as we shall present in Section 8.2.

8.2 Future Work

Although we consider the work presented in this dissertation is mature, complete and it offers solutions to some existing problems, it could still be improved in several directions. To begin with, we would like to study in more detail the expressiveness in the use of our observers, investigating which kinds of properties can be analyzed using observers and which ones cannot. We have shown that many properties for performance and reliability can be analyzed, so we would like to explore more QoS properties, such us safety, security and usability.

Since the simulation time of our systems in Maude increases as the number of model elements increases, we are planning to improve the internal representation of our models and of the observers in Maude. We would start by trying to decrease the number of conditions used in the conditional rules, since we have proved that the more conditions there are, the higher the simulation time. Since NACs are precisely included in this part of the conditional rules, we would like to study how detrimental the addition of NACs and the addition of action executions within them are for simulation times. If we conclude their addition is critical for this problem, we would think of a different representation for the NACs in Maude. We would, consequently, modify the ATL transformation from graphical behavioral rules into Maude.

The approach presented in this dissertation proposes to analyze non-functional properties of systems in the design phase. Thus, we are able to perform simulations in this phase. For this reason, we would like to explore the possible connection between the final implementation of the systems and their specifications in *e-Motions* in order to study the conformance of the system execution with its specified behavior. In this way, we could easily detect deviations from the expected behavior and degradations in the system performance with regards to its high-level visual specifications.

Regarding wireless sensor networks and our approach to model and analyze the DSAP and its reliability, we would like to define new variants apart from the ones presented in Chapter 4. In fact, since the modeling of the variants presented is very easy and direct in our implementation, we would like to seek a new variant that gives better reliability results. Likewise, we would like to explore new protocols for WSNs and model their reliability with our approach. Precisely for modeling and analyzing reliability in systems, we would like to study the inclusion of the bathtub curve (Figure 8.1) in our implementations. Such curve offers a view of the hazard function for system components. The hazard function is a measure of the tendency to fail: the greater the value of the hazard function, the greater the probability of impending failure. The curve represents the idea that the operation of a population of devices can be viewed as comprised of three distinct periods:

- an “early failure” (burn-in) period, where the hazard function decreases over time,

- a “random failure” (useful life) period, where the hazard function is constant over time,
- a “wear-out” period, where the hazard function increases over time.

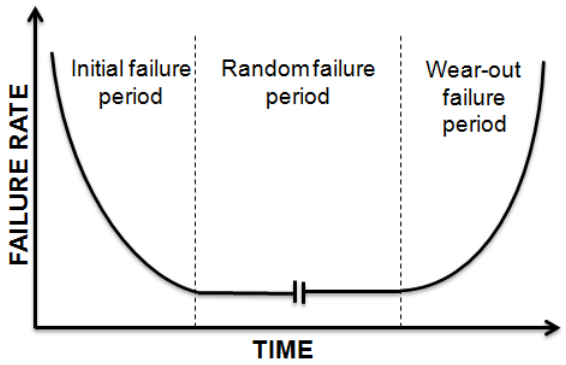


Figure 8.1: Bathtub Curve

We say we want to study whether to include the bathtub curve in our implementation or not because there are works where the authors present their refusal or disagreement with its use. For example, the work presented in [KKW03] exposes some of the limitations of the bathtub curve, and questions if any of the three segments of the curve models the reality in a fair way.

There is also room for improvement in our xQNM tool, since some new features can be added in new versions. For example, it could return, as result of the simulations, not the average of the different simulations, but a mixture of probability distributions (in case the behavior of the system is composed of several independent behaviors). We also plan on automatically distributing the simulations across several machines, by means of a concurrent and distributed solution that would use a task farm approach [Dan00], so that results would be collected faster. This idea, plus the improvement in the representation of models in Maude mentioned before, would speed up the simulations considerably. We could also consider to extend the behavior of our generic behavioral model for QNMs. For example, since we can take now failures in servers into account, we could include some rules for re-adapting the network when jobs are waiting in queues whose sever is inactive.

Although the approach presented in Chapter 6 for the modular definition of observers has been implemented and proved in different case studies, it is still in a naive and early phase and many things may need to be added, studied and improved. For example, we would like to provide a graphical interface so that the bindings can be defined graphically. Being able to trace lines from an element in a DSVL to its correspondent element in another DSVL would be of great help to the user. Right now, the bindings are realized by defining a correspondences model conforming to the metamodel shown in Fig. 6.8. We create that model with

the Eclipse tree view. We also need to study our approach for a wider variety of observers. Concretely, we have to study the case where there is no simple alignment between the rules of the observers and the systems. Furthermore, it would be ideal if some bindings could be realized automatically, with no human intervention. In this case, it could be necessary to detect pattern in rules and metamodels, instead of defining correspondences. In any case, the work needed to get a powerful approach from our first idea, implemented, tested, and presented in Chapter 6 might be huge, and the more we dig into it, the more challenges we will find. In fact, digging into the approach and getting a powerful solution could be the work of a master thesis or even a doctoral dissertation.

A

Probabilistic Distributions in Maude

This appendix shows the Maude implementation of the probabilistic distributions described in Section 3.4. Before presenting the code, we need to clarify something about the random numbers used. The formulae presented in Section 3.4 use a random number in the interval $[0,1]$. In Maude, the functional module RANDOM adds to NAT a pseudo-random number generator (Listing A.1).

Listing A.1: Maude RANDOM module

```
fmod RANDOM is
    protecting NAT .
    op random : Nat -> Nat [special (...)] .
endfm
```

The function random is the mapping from Nat into the range of natural numbers $[0, 2^{32}-1]$ computed by successive calls to the Mersenne Twister Random Number Generator [CDE+07]. Such random generator needs a seed, which by default is 0. In the transformation *e-Motions*–Maude, a different seed is generated whenever the random function available in *e-Motions* is used. Said seed is generated according to the current time and date of the transformation, making sure that every seed is different. As mentioned before, the function random available in Maude returns a natural number between 0 and 4294967295, and not a real number in the range $[0,1]$. For this reason, we have adapted the formulae presented in Section 3.4 to Maude's random number generator. For example, for the *Exponential* distribution, whose formula –the inversion method– is $exp(\lambda) = \frac{\ln(1-u)}{-\lambda}$, we have used random numbers between 1 and 1000. The actual formula we use is $x = \frac{-\ln(rnd(1,1000))+\ln(1000)}{\lambda}$.

In the following we present the Maude code that implements all the distributions. Note that we have considered that input parameters can be of types integer (Int), float (Float) and rational (PosRat), what allows *e-Motions* users to insert numbers of any type as parameters in their probability distributions. Note as well the presence of the last parameter in every function, which represents the seed to be used in the random number generator.

Listing A.2: Probability distribution functions defined in Maude

```
mod MGDISTRIBUTIONS is
  pr MAUDELING .
  pr MGRANDOM .
```

```
vars F L M SG SH SC ND DD P LB UB : Float .
   --- L: lambda, M: mu, SG: sigma [Varianze = sigma^2]
   --- SH: shape, SC: scale NN:numerator degree
   --- DD: denominator degree
   --- LB: lower bound, UB: upper bound
vars S LI MI SGI SHI SCI NDI DDI PI LBI UBI : Int .
   --- S: seed, LI: lambda-integer, MI: mu-integer
   --- SGI: sigma-integer, SHI: shape-integer
   --- SCI: scale-integer, NDI: numerator degree integer
   --- DDI: denominator degree integer
   --- LBI: lower bound integer, upper bound integer
vars LPR MPR SGPR SHPR SCPR NDPR DDPR PPR LBPR UBPR : PosRat .
   --- LPR: lambda-posrat MPR: mu-posrat
   ---SGPR: sigma-posrat SHPR: shape-posrat SCPR: scale-posrat
   ---NDPR: numerator degree posrat DDPR: denominator degree posrat
   ---LBPR: lower bound PosRat UBPR: upper bound PosRat
op round : Float -> Int .
eq round(F) = if F - floorF(F) >= 0.5 then ceilingF(F)
              else floorF(F) fi .

------------------EXPONENTIAL DISTRIBUTION-----------------------
op expDistr : Float Int -> Int .
eq expDistr (L, S) = rat(round((6.9077553 - log(rat2float(random(S)
                     rem 1000 + 1))) / L)) .
--- If lambda is an integer number instead of a float number:
op expDistr : Int Int -> Int .
eq expDistr (LI, S) = expDistr(rat2float(LI), S) .
--- If lambda is a PosRat number instead of a float number:
op expDistr : PosRat Int -> Int .
eq expDistr (LPR, S) = expDistr(rat2float(LPR), S) .

--- Auxiliar function to be used in the gamma distribution
op expFloat : Float Int -> Float .
eq expFloat (L, S) = (6.9077553 - log(rat2float(random(S)
                     rem 1000 + 1))) / L .
------------------------------------------------------------------
--------------------WEIBULL DISTRIBUTION-----------------------
--- M is the scale and SG is the shape
op weibDistr : Float Float Int -> Int .
eq weibDistr(M, SG, S) = rat(round((rat2float(-1.0 *F M) *F
                         log(rat2float((random(S) rem 1000 + 1) /
                         1000))) ^F (1.0 /F SG))) .
--- If mu and/or sigma are integer or posrat numbers instead of
--- float numbers:
op weibDistr : Int Int Int -> Int .
eq weibDistr(MI, SGI, S) = weibDistr(rat2float(MI),
                           rat2float(SGI), S) .
op weibDistr : Int Float Int -> Int .
eq weibDistr(MI, SG, S) = weibDistr(rat2float(MI), SG, S) .
op weibDistr : Int PosRat Int -> Int .
eq weibDistr(MI, SGPR, S) = weibDistr(rat2float(MI),
                            rat2float(SGPR), S) .
op weibDistr : Float Int Int -> Int .
eq weibDistr(M, SGI, S) = weibDistr(M, rat2float(SGI), S) .
op weibDistr : Float PosRat Int -> Int .
eq weibDistr(M, SGPR, S) = weibDistr(M, rat2float(SGPR), S) .
op weibDistr : PosRat Int Int -> Int .
eq weibDistr(MPR, SGI, S) = weibDistr(rat2float(MPR),
                            rat2float(SGI), S) .
op weibDistr : PosRat PosRat Int -> Int .
eq weibDistr(MPR, SGPR, S) = weibDistr(rat2float(MPR),
                             rat2float(SGPR), S) .
```

```
op weibDistr : PosRat Float Int -> Int .
eq weibDistr(MPR, SG, S) = weibDistr(rat2float(MPR), SG, S) .
------------------------------------------------------------
------------------NORMAL/GAUSS DISTRIBUTION-----------------
--- Standard normal distribution (mean = 0, varianze = 1):
op standNormDistr : Int -> Float .
eq standNormDistr(S) = sqrt(-2.0 *F log(rat2float((random(S)
                       rem 1000 + 1) / 1000))) *F
                       cos(6.283185307 *F rat2float(
                       (random(S + 1) rem 1000 + 1) / 1000)) .

op normDistr : Float Float Int -> Int .
eq normDistr (M, SG, S) = if rat(round(SG *F
                            standNormDistr(S) + M)) < 0 then 0
                          else rat(round(SG *F
                            standNormDistr(S) + M)) fi .
--- If mu and/or sigma are integer or posrat numbers instead
--- of float numbers:
op normDistr : Int Int Int -> Int .
eq normDistr (MI, SGI, S) = normDistr(rat2float(MI),
                            rat2float(SGI), S) .
op normDistr : Int Float Int -> Int .
eq normDistr (MI, SG, S) = normDistr(rat2float(MI), SG, S) .
op normDistr : Int PosRat Int -> Int .
eq normDistr (MI, SGPR, S) = normDistr(rat2float(MI),
                             rat2float(SGPR), S) .
op normDistr : Float Int Int -> Int .
eq normDistr (M, SGI, S) = normDistr(M, rat2float(SGI), S) .
op normDistr : Float PosRat Int -> Int .
eq normDistr (M, SGPR, S) = normDistr(M, rat2float(SGPR), S) .
op normDistr : PosRat Int Int -> Int .
eq normDistr (MPR, SGI, S) = normDistr(rat2float(MPR),
                             rat2float(SGI), S) .
op normDistr : PosRat PosRat Int -> Int .
eq normDistr (MPR, SGPR, S) = normDistr(rat2float(MPR),
                              rat2float(SGPR), S) .
op normDistr : PosRat Float Int -> Int .
eq normDistr (MPR, SG, S) = normDistr(rat2float(MPR), SG, S) .
------------------------------------------------------------
------------------GAMMA DISTRIBUTION----------------------------
op gammaDistr : Float Float Int -> Int .
eq gammaDistr(SH, SC, S) = gammaDistr(rat(round(SH)), SC, S) .
--- If shape and/or scale are integer or posrat numbers instead
--- of float numbers:
op gammaDistr : Int Int Int -> Int .
eq gammaDistr(SHI, SCI, S) = rat(round(gammaCalculation (SHI,
                             rat2float(SCI), S))) .
op gammaDistr : Int Float Int -> Int .
eq gammaDistr(SHI, SC, S) = rat(round(gammaCalculation
                            (SHI, SC, S))) .
op gammaDistr : Int PosRat Int -> Int .
eq gammaDistr(SHI, SCPR, S) = rat(round(gammaCalculation
                              (SHI, rat2float(SCPR), S))) .
op gammaDistr : Float Int Int -> Int .
eq gammaDistr(SH, SCI, S) = rat(round(gammaCalculation (rat
                            (round(SH)), rat2float(SCI), S))) .
op gammaDistr : Float PosRat Int -> Int .
eq gammaDistr(SH, SCPR, S) = rat(round(gammaCalculation (rat(
                             round(SH)), rat2float(SCPR), S))) .
op gammaDistr : PosRat Int Int -> Int .
eq gammaDistr(SHPR, SCI, S) = rat(round(gammaCalculation
                              (rat(round(rat2float(SHPR))),
```

```
                              rat2float(SCI), S))) .
op gammaDistr : PosRat PosRat Int -> Int .
eq gammaDistr(SHPR, SCPR, S) = rat(round(gammaCalculation
                               (rat(round(rat2float(SHPR))),
                               rat2float(SCPR), S))) .
op gammaDistr : PosRat Float Int -> Int .
eq gammaDistr(SHPR, SC, S) = rat(round(gammaCalculation
                            (rat(round(rat2float(SHPR))), SC, S))) .

--- Auxiliary function for the calculation of the gamma distribution
op gammaCalculation : Int Float Int -> Float .
eq gammaCalculation(SHI, SC, S) =
       if SHI == 0
       then 0.0
       else expFloat(SC, S) + gammaCalculation(SHI - 1, SC, S + SHI)
       fi .
------------------------------------------------------------------
-----------------------CHI-SQUARE DISTRIBUTION--------------------
op chiSDistr : Float Int -> Int .
eq chiSDistr(SH, S) = gammaDistr(rat(round(SH / 2.0)), 2, S) .
--- In case the parameter is integer or posrat instead of float:
op chiSDistr : Int Int -> Int .
eq chiSDistr(SHI, S) = gammaDistr(SHI / 2, 2, S) .
op chiSDistr : PosRat Int -> Int .
eq chiSDistr(SHPR, S) = gammaDistr(SHPR / 2, 2, S) .
--- To be used by the F distribution calculation
op chiSFloat : Float Int -> Float .
eq chiSFloat(SH, S) = gammaCalculation(rat(round(SH / 2.0)),
                      2.0, S) .
------------------------------------------------------------------
------------------------ERLANG DISTRIBUTION-----------------------
op erlangDistr : Int Int Int -> Int .
eq erlangDistr(SHI, SCI, S) = rat(round(gammaCalculation
                              (SHI, rat2float(SCI), S))) .
op erlangDistr : Int Float Int -> Int .
eq erlangDistr(SHI, SC, S) = rat(round(gammaCalculation
                             (SHI, SC, S))) .
op erlangDistr : Int PosRat Int -> Int .
eq erlangDistr(SHI, SCPR, S) = rat(round(gammaCalculation
                               (SHI, rat2float(SCPR), S))) .
------------------------------------------------------------------
-----------------------F DISTRIBUTION-----------------------------
--- nd: numerator degree of freedom, dd: denominator degree
--- of freedom
op fDistr : Float Float Int -> Int .
eq fDistr(ND, DD, S) = rat(round((chiSFloat(ND, S) / ND) /
                       (chiSFloat(DD, S) / DD))) .
op fDistr : Int Float Int -> Int .
eq fDistr(NDI, DD, S) = fDistr(rat2float(NDI), DD, S) .
op fDistr : PosRat Float Int -> Int .
eq fDistr(NDPR, DD, S) = fDistr(rat2float(NDPR), DD, S) .
op fDistr : Float Int Int -> Int .
eq fDistr(ND, DDI, S) = fDistr(ND, rat2float(DDI), S) .
op fDistr : Int Int Int -> Int .
eq fDistr(NDI, DDI, S) = fDistr(rat2float(NDI), rat2float(DDI), S) .
op fDistr : PosRat Int Int -> Int .
eq fDistr(NDPR, DDI, S) = fDistr(rat2float(NDPR),
                          rat2float(DDI), S) .
op fDistr : Float PosRat Int -> Int .
eq fDistr(ND, DDPR, S) = fDistr(ND, rat2float(DDPR), S) .
op fDistr : Int PosRat Int -> Int .
eq fDistr(NDI, DDPR, S) = fDistr(rat2float(NDI),
```

```
                          rat2float(DDPR), S) .
op fDistr : PosRat PosRat Int -> Int .
eq fDistr(NDPR, DDPR, S) = fDistr(rat2float(NDPR),
                           rat2float(DDPR), S) .
------------------------------------------------------------------
----------------------GEOMETRIC DISTRIBUTION----------------------
op geomDistr : Float Int -> Int .
eq geomDistr(L, S) = rat(ceilingF( log(rat2float((random(S) rem
                     100000 + 1) / 100000)) /F log(1.0 - L))) .
op geomDistr : Int Int -> Int .
eq geomDistr(LI, S) = geomDistr(rat2float(LI), S) .
op geomDistr : PosRat Int -> Int .
eq geomDistr(LPR, S) = geomDistr(rat2float(LPR), S) .
--- For the calculation of the Pascal function
op geomExp : Float Int -> Int .
eq geomExp(L, S) = log(rat2float((random(S) rem 100000 + 1) /
                   100000)) /F log(1.0 - L) .
------------------------------------------------------------------
-------------------LOG-NORMAL DISTRIBUTION-----------------------
--- M is the scale and SG is the log-shape
op logNormDistr : Float Float Int -> Int .
eq logNormDistr(M, SG, S) = expDistr(1 / normDistr(M, SG, S), S) .
op logNormDistr : Int Float Int -> Int .
eq logNormDistr(MI, SG, S) = logNormDistr(rat2float(MI), SG, S) .
op logNormDistr : PosRat Float Int -> Int .
eq logNormDistr(MPR, SG, S) = logNormDistr(rat2float(MPR), SG, S) .
op logNormDistr : Float Int Int -> Int .
eq logNormDistr(M, SGI, S) = logNormDistr(M, rat2float(SGI), S) .
op logNormDistr : Int Int Int -> Int .
eq logNormDistr(MI, SGI, S) = logNormDistr(rat2float(MI),
                              rat2float(SGI), S) .
op logNormDistr : PosRat Int Int -> Int .
eq logNormDistr(MPR, SGI, S) = logNormDistr(rat2float(MPR),
                               rat2float(SGI), S) .
op logNormDistr : Float PosRat Int -> Int .
eq logNormDistr(M, SGPR, S) = logNormDistr(M, rat2float(SGPR), S) .
op logNormDistr : Int PosRat Int -> Int .
eq logNormDistr(MI, SGPR, S) = logNormDistr(rat2float(MI),
                               rat2float(SGPR), S) .
op logNormDistr : PosRat PosRat Int -> Int .
eq logNormDistr(MPR, SGPR, S) = logNormDistr(rat2float(MPR),
                                rat2float(SGPR), S) .
------------------------------------------------------------------
----------------------PASCAL DISTRIBUTION-----------------------
op pascalDistr : Float Float Int -> Int .
eq pascalDistr(P, M, S) = rat(round(pascalSum(P,
                          rat(round(M)), S))) .
op pascalDistr : Float Int Int -> Int .
eq pascalDistr(P, MI, S) = pascalDistr(P, rat2float(MI), S) .
op pascalDistr : Float PosRat Int -> Int .
eq pascalDistr(P, MPR, S) = pascalDistr(P, rat2float(MPR), S) .
op pascalDistr : Int Float Int -> Int .
eq pascalDistr(PI, M, S) = pascalDistr(rat2float(PI), M, S) .
op pascalDistr : Int Int Int -> Int .
eq pascalDistr(PI, MI, S) = pascalDistr(rat2float(PI),
                            rat2float(MI), S) .
op pascalDistr : Int PosRat Int -> Int .
eq pascalDistr(PI, MPR, S) = pascalDistr(rat2float(PI),
                             rat2float(MPR), S) .
op pascalDistr : PosRat Float Int -> Int .
eq pascalDistr(PPR, M, S) = pascalDistr(rat2float(PPR), M, S) .
op pascalDistr : PosRat Int Int -> Int .
```

```
eq pascalDistr(PPR, MI, S) = pascalDistr(rat2float(PPR),
                             rat2float(MI), S) .
op pascalDistr : PosRat PosRat Int -> Int .
eq pascalDistr(PPR, MPR, S) = pascalDistr(rat2float(PPR),
                              rat2float(MPR), S) .

op pascalSum : Float Int Int -> Float .
eq pascalSum(P, 0, S) = 0.0 .
eq pascalSum(P, MI, S) = geomExp(P, S) +
                         pascalSum(P, MI - 1, S + MI) [owise] .
------------------------------------------------------------------
----------------------PARETO DISTRIBUTION-----------------------
op paretDistr : Float Float Int -> Int .
eq paretDistr(SH, SC, S) = rat(round(SC / eval(((random(S)
                           rem 100000 + 1) / 100000) .
                           pow(1 / SH)))) .
op paretDistr : Float Int Int -> Int .
eq paretDistr(SH, SCI, S) = paretDistr(SH, rat2float(SCI), S) .
op paretDistr : Float PosRat Int -> Int .
eq paretDistr(SH, SCPR, S) = paretDistr(SH, rat2float(SCPR), S) .
op paretDistr : Int Float Int -> Int .
eq paretDistr(SHI, SC, S) = paretDistr(rat2float(SHI), SC, S) .
op paretDistr : Int Int Int -> Int .
eq paretDistr(SHI, SCI, S) = paretDistr(rat2float(SHI),
                             rat2float(SCI), S) .
op paretDistr : Int PosRat Int -> Int .
eq paretDistr(SHI, SCPR, S) = paretDistr(rat2float(SHI),
                              rat2float(SCPR), S) .
op paretDistr : PosRat Float Int -> Int .
eq paretDistr(SHPR, SC, S) = paretDistr(rat2float(SHPR), SC, S) .
op paretDistr : PosRat Int Int -> Int .
eq paretDistr(SHPR, SCI, S) = paretDistr(rat2float(SHPR),
                              rat2float(SCI), S) .
op paretDistr : PosRat PosRat Int -> Int .
eq paretDistr(SHPR, SCPR, S) = paretDistr(rat2float(SHPR),
                               rat2float(SCPR), S) .
------------------------------------------------------------------
----------------------UNIFORM DISTRIBUTION----------------------
op unifDistr : Float Float Int -> Int .
eq unifDistr(LB, UB, S) = LB + (UB - LB) *F rat2float((random(S)
                          rem 100000 + 1) / 100000) .
op unifDistr : Float Int Int -> Int .
eq unifDistr(LB, UBI, S) = unifDistr(LB, rat2float(UBI), S ) .
op unifDistr : Float PosRat Int -> Int .
eq unifDistr(LB, UBPR, S) = unifDistr(LB, rat2float(UBPR), S ) .
op unifDistr : Int Float Int -> Int .
eq unifDistr(LBI, UB, S) = unifDistr(rat2float(LBI), UB, S ) .
op unifDistr : Int Int Int -> Int .
eq unifDistr(LBI, UBI, S) = unifDistr(rat2float(LBI),
                            rat2float(UBI), S ) .
op unifDistr : Int PosRat Int -> Int .
eq unifDistr(LBI, UBPR, S) = unifDistr(rat2float(LBI),
                             rat2float(UBPR), S ) .
op unifDistr : PosRat Float Int -> Int .
eq unifDistr(LBPR, UB, S) = unifDistr(rat2float(LBPR), UB, S ) .
op unifDistr : PosRat Int Int -> Int .
eq unifDistr(LBPR, UBI, S) = unifDistr(rat2float(LBPR),
                             rat2float(UBI), S ) .
op unifDistr : PosRat PosRat Int -> Int .
eq unifDistr(LBPR, UBPR, S) = unifDistr(rat2float(LBPR),
                              rat2float(UBPR), S ) .
------------------------------------------------------------------
```

B

ATL Transformations for weaving DSLs

This appendix shows the implementation of the ATL transformations explained in Section 6.2. Such transformations aim to weave different DSLs in one. More precisely, they are used for incorporating non-functional concepts, expressed in observer DSLs, into specific systems. The schema of both transformations is shown in Figure 6.9. The first transformation, *WeaveMetaModels.atl*, weaves abstract and concrete syntaxes into one, while the second transformation, *WeaveBeh.atl*, weaves the behavioral rules.

The implementation of the *WeaveMetaModels.atl* transformation is the following.

Listing B.1: Implementation of the WeaveMetaModels.atl transformation

```
--- Transformation developed with Eclipse INDIGO

--- We have enabled "Inter-model References" in this transformation,
--- since our behavior models reference to GCS and ECORE

module WeaveMetaModels; -- Module Template
create  OUTMM : WeaveMM, OUTGCS : WeaveGCS
from INDSLMM : DSLMM, INDSLGCS : DSLGCS, INObsMM : ObsMM,
     INObsGCS : ObsGCS, INCorresp : CorrespMM ;

-- OUTMM : WeaveMM --> Metamodel resulting of the weave between
-- INDSLMM:DSLMM and INObsMM:ObsMM. It conforms to Ecore

-- OUTGCS : WeaveGCS --> Metamodel resulting of the weave between
-- INDSLGCS:DSLGCS and INObsGCS:ObsGCS. It conforms to the
-- GCS metamodel

-- INDSLMM : DSLMM --> Metamodel of the DSL. It conforms to Ecore

-- INDSLGCS : DSLGCS --> GCS model of the DSL. It conforms
-- to the GCS metamodel

-- INObsMM : ObsMM --> Metamodel of observers. It conforms to Ecore

-- INObsGCS : ObsGCS --> GCS model of the observers. It conforms
-- to the GCS metamodel

-- INCorresp : CorrespMM --> Model with the correspondences.
-- It conforms to the CorrespondencesMM metamodel

-- It will keep the package of the output metamodel:
helper def : packageMM : WeaveMM!EPackage = OclUndefined;
```

```
-- It will keep the package of the output GCS:
helper def : packageGCS : WeaveGCS!PackageGD = OclUndefined;

-- It will store all the classes created in the output metamodel:
helper def : classes : Sequence(WeaveMM!EClass) = Sequence{};

-------------1---RULES FOR COPYING THE DSL METAMODEL-----------------
rule CopyPackagesMM{
  from
    p : DSLMM!EPackage
  to
    pMM : WeaveMM!EPackage(
      name <- p.name,
      nsURI <- p.nsURI,
      nsPrefix <- p.nsPrefix,
      eSubpackages <- p.eSubpackages,
      eClassifiers <- p.eClassifiers,
      eAnnotations <- p.eAnnotations
    )
    do{
      thisModule.packageMM <- pMM;
    }
}

rule CopyClasses{
  from
    c : DSLMM!EClass
  to
    c2 : WeaveMM!EClass(
      instanceClassName <- c.instanceClassName,
      name <- c.name,
      abstract <- c.abstract,
      interface <- c.interface,
      eSuperTypes <- c.eSuperTypes,
      eOperations <- c.eOperations,
      eStructuralFeatures <- c.eStructuralFeatures,
      eAnnotations <- c.eAnnotations
    )
    do{
      --We insert the class inside our EPackage
      thisModule.packageMM.eClassifiers <-
                thisModule.packageMM.eClassifiers -> append(c2);
      --We add the new class to the set of classes
      thisModule.classes <- thisModule.classes -> including(c2);
    }
}

rule CopyAttributes{
  from
    a : DSLMM!EAttribute
  to
    a2 : WeaveMM!EAttribute(
      eType <- a.eType,
      name <- a.name,
      iD <- a.iD,
      changeable <- a.changeable,
      volatile <- a.volatile,
      transient <- a.transient,
      defaultValueLiteral <- a.defaultValueLiteral,
      unsettable <- a.unsettable,
      derived <- a.derived,
```

```
      ordered <- a.ordered,
      unique <- a.unique,
      lowerBound <- a.lowerBound,
      upperBound <- a.upperBound,
      eAnnotations <- a.eAnnotations
    )
}

rule CopyReferences{
  from
    r : DSLMM!EReference
  to
    r2 : WeaveMM!EReference(
      name <- r.name,
      containment <- r.containment,
      eOpposite <- r.eOpposite,
      changeable <- r.changeable,
      volatile <- r.volatile,
      transient <- r.transient,
      defaultValueLiteral <- r.defaultValueLiteral,
      unsettable <- r.unsettable,
      derived <- r.derived,
      ordered <- r.ordered,
      unique <- r.unique,
      lowerBound <- r.lowerBound,
      upperBound <- r.upperBound,
      eType <- r.eType,
      eAnnotations <- r.eAnnotations
    )
}

rule CopyDataTypes{
  from
    dt : DSLMM!EDataType
  to
    dt2 : WeaveMM!EDataType(
      serializable <- dt.serializable,
      instanceClassName <- dt.instanceClassName,
      name <- dt.name,
      eAnnotations <- dt.eAnnotations
    )
}

rule CopyParameters{
  from
    p : DSLMM!EParameter
  to
    p2 : WeaveMM!EParameter(
      name <- p.name,
      eOperation <- p.eOperation,
      ordered <- p.ordered,
      unique <- p.unique,
      lowerBound <- p.lowerBound,
      upperBound <- p.upperBound,
      eType <- p.eType,
      eAnnotations <- p.eAnnotations
    )
}

rule CopyOperations{
  from
    o : DSLMM!EOperation
```

```
  to
    o2 : WeaveMM!EOperation(
      name <- o.name,
      eContainingClass <- o.eContainingClass,
      eParameters <- o.eParameters,
      eExceptions <- o.eExceptions,
      ordered <- o.ordered,
      unique <- o.unique,
      lowerBound <- o.lowerBound,
      upperBound <- o.upperBound,
      eType <- o.eType,
      eAnnotations <- o.eAnnotations
    )
}

rule CopyEnums{
  from
    e : DSLMM!EEnum
  to
    e2 : WeaveMM!EEnum(
      name <- e.name,
      eLiterals <- e.eLiterals,
      instanceClassName <- e.instanceClassName,
      instanceClass <- e.instanceClass,
      defaultValue <- e.defaultValue,
      eAnnotations <- e.eAnnotations
    )
}

rule CopyEnumLiterals{
  from
    e : DSLMM!EEnumLiteral
  to
    e2 : WeaveMM!EEnumLiteral(
      name <- e.name,
      value <- e.value,
      instance <- e.instance,
      eEnum <- e.eEnum,
      eAnnotations <- e.eAnnotations
    )
}

rule CopyAnnotations{
  from
    a : DSLMM!EAnnotation
  to
    a2 : WeaveMM!EAnnotation(
      source <- a.source,
      details <- a.details,
      eAnnotations <- a.eAnnotations
    )
}

rule CopyFactories{
  from
    f : DSLMM!EFactory
  to
    f2 : WeaveMM!EFactory(
      ePackage <- f.ePackage,
      eAnnotations <- f.eAnnotations
    )
}
```

```
------------END OF RULES FOR COPYING THE DSL METAMODEL--------------

------------2---RULES FOR COPYING THE DSL AND OBSERVER GCS-----------
rule CopyPackagesGCS{
  from
    p : DSLGCS!PackageGD
  to
    pGCS : WeaveGCS!PackageGD(
      name <- p.name,
      metamodelGD <- mmGCS,
      superPackageGD <- p.superPackageGD,
      subPackagesGD <- p.subPackagesGD,
      package <- p.package
    ),
    mmGCS : WeaveGCS!MetamodelGD(
      name <- p.metamodelGD.name
    )
  do{
    thisModule.packageGCS <- pGCS;
  }
}

rule CopyClassesGDDSL{
  from
    c : DSLGCS!ClassGD
  to
    c2 : WeaveGCS!ClassGD(
      name <- c.name,
      figurePath <- c.figurePath,
      class <- c.class
    )
  do{
    --We insert the class inside our EPackage
    thisModule.packageGCS.classesGD <-
            thisModule.packageGCS.classesGD -> append(c2);
  }
}

rule CopyClassesGDObs{
  from
        -- We want to add the ClassGD only from the observers,
        -- not from those classes having a correspondence
         -- (observers do not have correspondences)
    c : ObsGCS!ClassGD,
    cObs : ObsMM!EClass(cObs.name=c.name and not
            CorrespMM!ClassMatching.allInstances() ->
            exists(cm | cm.obName = c.name))
  to
    c2 : WeaveGCS!ClassGD(
      name <- c.name,
      figurePath <- c.figurePath,
      class <- thisModule.CreateNonFunctionalClass(cObs)
    )
  do{
    --We insert the class inside our EPackage
    thisModule.packageGCS.classesGD <-
            thisModule.packageGCS.classesGD -> append(c2);
  }
}
---------END OF RULES FOR COPYING THE DSL AND OBSERVER GCS------------

---------3---RULES FOR CREATING REFERENCES, CLASSES AND --------------
```

```
--------------ATTRIBUTES OF NONFUNCTIONAL PROPERTIES----------------
rule CreateObserverOnlyReference {
-- This rule aims to include in the final metamodel a reference (and
-- class) that is included in the observers metamodel but not in the
-- DSL metamodel. To achieve it, the rule will look in the
-- CorrespondenceMM for two classes that match. The condition to
-- trigger the rule is also that it finds a reference departing
-- from the class in the matching belonging to the observers metamodel
-- such as there is not a matching between that reference and any
-- reference in the DSL metamodel. The rule will create that reference
-- departing from (the class in the WeaveMM model created from) the
-- class in the DSL metamodel as well as the class at the other end
-- of the reference.
  from
    cc : CorrespMM!ClassMatching, -- Correspondence between the
                                  -- observer and DSL classes
    cDSL : DSLMM!EClass, -- Class in the correspondence that
                         -- belongs to the DSL metamodel
    rObs : ObsMM!EReference, -- Reference departing from a class
                             -- in the observer metamodel. This
                             -- reference does not have a match
                             -- in the correspondence metamodel
    cEndRefObs : ObsMM!EClass, -- This is the class at the other
                               -- end of the rObs reference,
                               -- i.e.,the one representing the
                               -- non-functional property
    -- The next one is the class in the correspondence that
    -- belongs to the observer metamodel. We check that the
    -- rObs reference does not have a match with any reference
    -- in the DSL metamodel.
    cObs : ObsMM!EClass (cObs.name = cc.obName and cDSL.name =
        cc.DSLName and cObs.eAllReferences->includes(rObs) and
        rObs.eReferenceType = cEndRefObs and
        CorrespMM!RefMatching.allInstances() -> select(rm |
        rm.obClassName=cObs.name and rm.obRefName=rObs.name) ->
        size() = 0
    )
  to
    rDSL : WeaveMM!EReference( -- The new reference pointing the class
                               -- representing the non-Functional
                               -- property
      name <- rObs.name,
      -- We create the class pointed by this reference by means of a
        -- unique lazy rule, because it can happen that two classes
        -- in the observer metamodel reference the same nonFunctional
        -- class. In such case, the nonFunctional class has to
        -- be created only once
      eType <- thisModule.CreateNonFunctionalClass(cEndRefObs),
      containment <- rObs.containment,
      changeable <- rObs.changeable,
      volatile <- rObs.volatile,
      transient <- rObs.transient,
      defaultValueLiteral <- rObs.defaultValueLiteral,
      unsettable <- rObs.unsettable,
      derived <- rObs.derived,
      ordered <- rObs.ordered,
      unique <- rObs.unique,
      lowerBound <- rObs.lowerBound,
      upperBound <- rObs.upperBound,
      eAnnotations <- rObs.eAnnotations
    )
    do{
```

```
      -- We have to stablish the source class for the new reference
      -- created. To do that, we cannot use the eContainingClass
      -- of the reference because that property is non-changeable,
      -- so we look in the set of classes created in order to find
      -- the one with the same name as cDSL, and then we use its
      -- eStructuralFeatures reference
      for (c in thisModule.classes){
        if (c.name = cDSL.name){
          c.eStructuralFeatures <- c.eStructuralFeatures
                  -> append(rDSL);
        }
      }
    }
}


unique lazy rule CreateNonFunctionalClass{
-- This rule creates those classes representing non-functional
-- properties. It is a unique lazy rule because two different
-- classes in the obs metamodel may be referencing the same
-- non-functional class. In such case, this rule will be called
-- more than once, but only one class has to be created
  from
    cEndRefObs : ObsMM!EClass  -- This is the class at the other end
                               -- of the rObs reference, i.e., the one
                               -- representing the non-functional
                               -- property
  to
    cEndRefDSL : WeaveMM!EClass( -- The new class with a non-
                                 -- functional property to be added
                -- in the output metamodel
      instanceClassName <- cEndRefObs.instanceClassName,
      name <- cEndRefObs.name,
      abstract <- cEndRefObs.abstract,
      interface <- cEndRefObs.interface,
      eOperations <- cEndRefObs.eOperations,
       -- It will look for the EAttributes created by the
       -- "CreateNonFuncionalAttribute" rule:
      eStructuralFeatures <- cEndRefObs.eStructuralFeatures,
      eAnnotations <- cEndRefObs.eAnnotations
    )
  do{
    -- We add the new created class in the package
    thisModule.packageMM.eClassifiers <- thisModule.packageMM.
      eClassifiers -> append(cEndRefDSL);
    --We add the new class to the set of classes
    thisModule.classes <- thisModule.classes -> including(cEndRefDSL);
  }
}

rule CreateNonFunctionalAttribute{
-- This rule creates the attributes for those classes representing
-- non-functional properties. Those classes do not have a
-- correspondence in the correspondence model (that is how we identify
-- them). The attributes created by this rule are to be contained by
-- the class created by the unique lazy rule CreateNonFunctionalClass
-- (eStructuralFeatures <- cEndRefObs.eStructuralFeatures). That rule
-- will look in the traceability links for those attributes belonging
-- to the ObsMM!EClass in the LHS of the rule
  from
    a : ObsMM!EAttribute (not CorrespMM!ClassMatching.allInstances()->
          exists(cm|cm.obName=a.eContainingClass.name))
```

```
  to
    a2 : WeaveMM!EAttribute(
      eType <- a.eType,
      name <- a.name,
      iD <- a.iD,
      changeable <- a.changeable,
      volatile <- a.volatile,
      transient <- a.transient,
      defaultValueLiteral <- a.defaultValueLiteral,
      unsettable <- a.unsettable,
      derived <- a.derived,
      ordered <- a.ordered,
      unique <- a.unique,
      lowerBound <- a.lowerBound,
      upperBound <- a.upperBound,
      eAnnotations <- a.eAnnotations
    )
}
-----------------------END OF TRANSFORMATION-------------------------
```

The implementation of the *WeaveBeh.atl* transformation is shown below.

Listing B.2: Implementation of the WeaveBeh.atl transformation

```
-- Transformation developed with Eclipse INDIGO
-- We have enabled "Inter-model References" in this transformations,
-- since our behavior models reference to GCS and ECORE

module WeaveBeh; -- Module Template
create  OUT : WeaveBeh from INDSLBeh : DSLBeh,  INObsBeh : ObsBeh,
    INWeaveGCS : WeaveGCS, INWeaveMM : WeaveMM, INCorresp : CorrespMM;

-- OUT : WeaveBeh --> Behavior rules resulting of the weave between
-- INDSLBeh:DSLBeh and  INObsBeh:ObsBeh.  It conforms to the Behavior
-- metamodel

-- INDSLBeh : DSLBeh --> Behavioral rules  of the DSL. It conforms
-- to the Behavior metamodel

-- INObsBeh : ObsBeh --> Behavioral rules of observers. It conforms
-- to the Behavior metamodel

-- INWeaveGCS : WeaveGCS --> GCS model of the weave metamodel result
-- of the WeaveMetamodels.atl transformation. It conforms to the
-- GCS metamodel.

-- INWeaveMM : WeaveMM --> Weave Metamodel result of the
-- WeaveMetamodels.atl transformation. It conforms to Ecore

-- INCorresp : CorrespMM --> Model with the correspondences.
-- It conforms to the CorrespondencesMM metamodel

helper def : slot : WeaveBeh!Slot = OclUndefined;
helper def : beh : WeaveBeh!Behavior = OclUndefined;
-- Variable for keeping the rules that we create in the weaving
-- model:
helper def : rules : Set(WeaveBeh!Rule) = Set{};

-- Given a  CorrespMM!RuleMatching, a DSLBeh!Pattern (the pattern in
-- a rule) and a ObsBeh!Link, which is the link in the pattern (either
-- lhs or rhs) of a rule going from an object (which has a
-- correspondence object in the DSLBeh) to another object representing
```

```
-- a non-functional property, this helper returns the object that
-- corresponds to the source object of the link:
helper def: getDSLObjLHS(l : ObsBeh!Link, rm : CorrespMM!RuleMatching,
    pDSL : DSLBeh!Pattern) : DSLBeh!Object =
  pDSL.els->select(o|o.oclIsKindOf(DSLBeh!Object))->select(o|o.id =
      thisModule.getDSLObjIdLHS(rm, l.src))->first()
;

-- Given a CorrespondenceMM!RuleMatching and an ObsBeh!Object,
-- this helper returns the identifier of the DSLBeh!Object that
-- corresponds to the ObsBeh!Object:
helper def: getDSLObjIdLHS(rm : CorrespMM!RuleMatching, oOb :
    ObsBeh!Object) : String =
  rm.lhsObjs->select(om|om.obName = oOb.id)->first().DSLName
;

-------1---RULES FOR COPYING THE BEHAVIORAL RULES OF THE DSL----------
entrypoint rule Initialize(){
  to
    b : WeaveBeh!Behavior(
      denseTime <- false,
      nonInjectiveness <- false,
      metamodelGD <- WeaveGCS!MetamodelGD.allInstances() -> first()
    )
  do{
    thisModule.beh <- b; -- We keep our woven behavior model
                         -- in this variable
  }
}

rule CopyAtomicRules{
  from
    r : DSLBeh!AtomicRule
  to
    outR : WeaveBeh!AtomicRule(
      name <- r.name,
      lowerBound <- r.lowerBound,
      upperBound <- r.upperBound,
      soft <- r.soft,
      vbles <- r.vbles,
      lhs <- r.lhs,
      rhs <- r.rhs,
      nacs <- r.nacs,
      behavior <- thisModule.beh,
      patterns <- r.patterns,
      maxDuration <- r.maxDuration,
      minDuration <- r.minDuration
    )
  do{
    thisModule.rules <- thisModule.rules -> including (outR);
  }
}

rule CopyOngoingRules{
  from
    r : DSLBeh!OngoingRule
  to
    outR : WeaveBeh!OngoingRule(
      name <- r.name,
      lowerBound <- r.lowerBound,
      upperBound <- r.upperBound,
      soft <- r.soft,
```

```
      vbles <- r.vbles,
      lhs <- r.lhs,
      rhs <- r.rhs,
      nacs <- r.nacs,
      behavior <- thisModule.beh,
      patterns <- r.patterns,
      maxDuration <- r.maxDuration
    )
  do{
    thisModule.rules <- thisModule.rules -> including (outR);
  }
}

rule CopyHelpers{
  from
    h : DSLBeh!Helper
  to
    outH : WeaveBeh!Helper(
      oclExpression <- h.oclExpression,
      behavior <- thisModule.beh
    )
}

rule CopyVariables{
  from
    v : DSLBeh!Variable
  to
    outV : WeaveBeh!Variable(
      name <- v.name,
      type <- v.type,
      value <- v.value,
      "rule" <- v."rule"
    )
}

rule CopyPatterns{
  from
    p : DSLBeh!Pattern
  to
    outP : WeaveBeh!Pattern(
      name <- p.name,
      "rule" <- p."rule",
      els <- p.els
    )
}

rule CopyActionExecs{
  from
    a : DSLBeh!ActionExec
  to
    outA : WeaveBeh!ActionExec(
      action <- a.action,
      participants <- a.participants,
      id <- a.id,
      startingTime <- a.startingTime,
      endingTime <- a.endingTime,
      status <- a.status,
      executionTime <- a.executionTime
    )
}

rule CopyObjectRoles{
```

```
  from
    o : DSLBeh!ObjectRole
  to
    outO : WeaveBeh!ObjectRole(
      actualObject <- o.actualObject,
      role <- o.role,
      actionExec <- o.actionExec
    )
}

rule CopyObjects{
  from
    o : DSLBeh!Object(not o.oclIsKindOf(DSLBeh!Clock))
  to
    outO : WeaveBeh!Object(
      id <- o.id,
      sfs <- o.sfs,
      -- The class of the object has to be that in the WeaveMM
      -- and not that in the DSLMM
      classGD <- WeaveGCS!ClassGD.allInstances()->
                 select(c|c.name=o.classGD.name)->first(),
      outLinks <- o.outLinks,
      inLinks <- o.inLinks
    )
}

rule CopyClocks{
  from
    c : DSLBeh!Clock
  to
    outC : WeaveBeh!Clock(
      id <- c.id,
      sfs <- c.sfs,
      classGD <- c.classGD,
      outLinks <- c.outLinks,
      inLinks <- c.inLinks,
      pattern <- c.pattern
    )
}

rule CopySlots{
  from
    s : DSLBeh!Slot
  to
    outS : WeaveBeh!Slot(
      -- sf must point to the EAttribute in the WeaveMM
      -- metamodel. In fact, in can also be a reference (like in the
      -- Collect rule). This is why we will with this in the
      -- imperative part
      object <- s.object,
      oclValue <- s.oclValue
    )
  do{
    -- The slot can represent an EAttribute or an EReference, so
    -- we have to check it
    -- Here, we simply use WeaveMM to refer to the Ecore metamodel:
    if (s.sf.oclIsKindOf(WeaveMM!EAttribute)){
      outS.sf <- WeaveMM!EAttribute.allInstances()->
            select(r|r.name=s.sf.name and r.eContainingClass.name=
            s.sf.eContainingClass.name)->first();
    } else { -- If the s.sf is a reference
      outS.sf <- WeaveMM!EReference.allInstances()->
```

```
            select(r|r.name=s.sf.name and r.eContainingClass.name=
            s.sf.eContainingClass.name)->first();
    }
  }
}


rule CopyLinks{
  from
    l : DSLBeh!Link
  using{
    -- Class in the WeaveMM to which the source of the link is
    -- type of
    c : WeaveMM!EClass = WeaveMM!EClass.allInstances()->
        select(cl|cl.name=l.src.classGD.name)->first();
  }
  to
    outL : WeaveBeh!Link(
      src <- l.src, -- This is an Object in Behavior
                    -- (l.src.classGD is the class and
                    -- l.src.id is the id)
      target <- l.target,
      -- ref.name is the name of the EReference (the link).
      -- It has to be a EReference of the input WeaveMM
      -- We look for that EReference whose name is the same as
      -- the name of l.ref and whose source class is the same as
      -- l.src or a super type of it
      ref <- WeaveMM!EReference.allInstances()->
            select(r|r.name=l.ref.name and (c.eStructuralFeatures->
            exists(r|r.oclIsKindOf(WeaveMM!EReference) and
            r.name = l.ref.name) or c.eAllSuperTypes->
            exists(cl|cl.eStructuralFeatures->
            exists(r|r.oclIsKindOf(WeaveMM!EReference) and
            r.name=l.ref.name))))->first(),
      pos <- l.pos
    )
}

rule CopyConditions{
  from
    c : DSLBeh!Condition
  to
    outC : WeaveBeh!Condition(
      oclValue <- c.oclValue
    )
}
-------END OF RULES FOR COPYING THE BEHAVIORAL RULES OF THE DSL-------

-----------2---RULES FOR INCLUDING OBSERVER OBJECTS AND---------------
-------------LINKS IN THOSE RULES HAVING CORRESPONDENCES--------------
rule CreateObserversLHS {
-- This rule aims to include in the LHS of the DSL rules that
-- correspond with observer rules an object and link that are included
-- in the observers rule but not in the DSL rule. To achieve it, the
-- rule looks in the CorrespondenceMM for two rules that match. The
-- condition to trigger the rule is also to find a link in the LHS of
-- an observer rule that does not have a correspondent link, which
-- means that it is pointing to an object representing a
-- non-functional property. This link, as well as the object
-- representing the non functional property, has to be included in
-- the merged behavioral rule. This matched rule will create that
-- reference departing from (the class in the WeaveMM model created
```

```
-- from) the class in the DSL metamodel as well as the class at
-- the other end of the reference.
  from
    -- Correspondence between an observer and DSL rules:
    rm : CorrespMM!RuleMatching,
    -- Correspondence between the objects in the rules:
    om : CorrespMM!ObjectMatching,
    rObs : ObsBeh!Rule, -- Rule of the observer
    rDSL : DSLBeh!Rule, -- Rule of the DSL

    oDSL : DSLBeh!Object, -- Object in the correspondence that
                          -- belongs to the DSL metamodel
    -- Link departing from an object in the observer rule. This
    -- link does not have a match in the correspondence model:
    lObs : ObsBeh!Link,
    -- This is the object at the other end of the lObs link,
    -- i.e., the one representing the non-functional property:
    oEndLinkObs : ObsBeh!Object,
    -- The next one is the object in the correspondence that
    -- belongs to the observer rule. We check that the lObs link
    -- does not have a match with any link in the DSL metamodel.
    -- We check as well that the objects are included in the
    --LHS of the rules doing the matching in the rm:
    oObs : ObsBeh!Object (rObs.lhs.els->includes(oObs) and
        rDSL.lhs.els->includes(oDSL) and rObs.name=rm.obRuleName
        and rDSL.name=rm.DSLRuleName and rm.lhsObjs->includes(om)
        and oObs.id = om.obName and oDSL.id = om.DSLName and
        oObs.outLinks->includes(lObs) and lObs.src = oObs and
        lObs.target = oEndLinkObs and rm.lhsLinks ->
        select (lm|lm.obObjName=oObs.id
        and lm.obLinkName=lObs.ref.name) -> size() = 0)
  using{
    -- Class in the WeaveMM (class originally belonging to
    -- DSL) to which the source of the lObs link (class
    -- originally belonging to Obs) corresponds to
    c : WeaveMM!EClass = WeaveMM!EClass.allInstances()->
        select(cl|cl.name=oDSL.classGD.name)->first();
  }
  to
    -- The new link pointing the object representing the
    -- non-functional property:
    lNonFunc : WeaveBeh!Link(
      -- We look for the appropriate EReference in the
      -- WeaveMM metamodel
      ref <- WeaveMM!EReference.allInstances()->
            select(r|r.name=lObs.ref.name and
            (c.name=r.eContainingClass.name or c.eAllSuperTypes->
            exists(cl|cl.name=r.eContainingClass.name)))->first(),
      -- The source object of the link has to be the object
      -- in the DSL rule, i.e., the object created by the
      -- "CopyObjects" rule from that object:
      src <- oDSL,
      target <- oNonFunc,
      pos <-lObs.pos,
      pattern <- rDSL.lhs
    ),
    oNonFunc : WeaveBeh!Object(
      id <- oEndLinkObs.id,
      sfs <- oEndLinkObs.sfs -> collect(s |
            thisModule.CreateSlot(s, oNonFunc)),
      -- The class of the object has to be that in the
      -- WeaveMM and not that in the ObsMM
```

```
      classGD <- WeaveGCS!ClassGD.allInstances()->
            select(c|c.name=oEndLinkObs.classGD.name)->first(),
      inLinks <- Sequence{lNonFunc},
      pattern <- rDSL.lhs
    )
}

rule CreateObserversRHS {
-- This rule does the same as the previous one but with the RHS
  from
    rm : CorrespMM!RuleMatching,
    om : CorrespMM!ObjectMatching,
    rObs : ObsBeh!Rule,
    rDSL : DSLBeh!Rule,
    oDSL : DSLBeh!Object,
    lObs : ObsBeh!Link,
    oEndLinkObs : ObsBeh!Object,
    oObs : ObsBeh!Object (rObs.rhs.els->includes(oObs) and
        rDSL.rhs.els->includes(oDSL) and rObs.name=rm.obRuleName and
        rDSL.name=rm.DSLRuleName and rm.rhsObjs->includes(om) and
        oObs.id = om.obName and oDSL.id =
        om.DSLName and oObs.outLinks-> includes(lObs) and lObs.src =
        oObs and lObs.target = oEndLinkObs and rm.rhsLinks -> select
        (lm|lm.obObjName=oObs.id and
        lm.obLinkName=lObs.ref.name) -> size() = 0)
  using{
    c : WeaveMM!EClass = WeaveMM!EClass.allInstances()->
        select(cl|cl.name=oDSL.classGD.name)->first();
  }
  to
    lNonFunc : WeaveBeh!Link(
      ref <- WeaveMM!EReference.allInstances()->select
            (r|r.name=lObs.ref.name and
            (c.name=r.eContainingClass.name or
            c.eAllSuperTypes->exists(cl|cl.name=r.
            eContainingClass.name)))->first(),
      src <- oDSL,
      target <- oNonFunc,
      pos <-lObs.pos,
      pattern <- rDSL.rhs
    ),
    oNonFunc : WeaveBeh!Object(
      id <- oEndLinkObs.id,
      sfs <- oEndLinkObs.sfs -> collect(s |
            thisModule.CreateSlot(s, oNonFunc)),
      classGD <- WeaveGCS!ClassGD.allInstances()->
            select(c|c.name=oEndLinkObs.classGD.name)->first(),
      inLinks <- Sequence{lNonFunc},
      pattern <- rDSL.rhs
    )
}

lazy rule CreateSlot{
  from
    s : ObsBeh!Slot,
    oNonFunc : WeaveBeh!Object
  to
    outS : WeaveBeh!Slot(
      -- sf must point to the EAttribute in the WeaveMM
      -- metamodel. In fact, in can also be a reference (like in
      -- the Collect rule). That is why we will with this in the
      -- imperative part
```

```
      object <- oNonFunc,
      -- If the value of the slot in the observer object is "T",
      -- it means that it has to access the time elapsed in the DSL,
      -- which we identify as "clk.time" - the object clock is
      -- idenfified as "clk"
      oclValue <- if s.oclValue='T' then 'clk.time'
            else s.oclValue endif
    )
  do{
    -- Here, we simple use WeaveMM to refer to the Ecore metamodel:
    if (s.sf.oclIsKindOf(WeaveMM!EAttribute)){
      outS.sf <- WeaveMM!EAttribute.allInstances()->
            select(r|r.name=s.sf.name and
            r.eContainingClass.name=s.sf.eContainingClass.name)
            ->first();
    } else { -- If the s.sf is a reference
      outS.sf <- WeaveMM!EReference.allInstances()->
            select(r|r.name=s.sf.name and r.eContainingClass.name=
            s.sf.eContainingClass.name)->first();
    }
  }
}

---------END OF RULES FOR INCLUDING OBSERVER OBJECTS AND--------------
-------------LINKS IN THOSE RULES HAVING CORRESPONDENCES--------------

---------3---RULES FOR INCLUDING CLOCKS IN THE WOVEN RULES------------
-- Since the majority of the observers require using the time elapsed
-- in the DSL, we add the CLOCK object in all those rules where this
-- CLOCK object was not included before
rule CreateClock {
-- We suppose that the DSL rules are well formed, i.e., if there is
-- an object clock in the LHS of the rule, then it is also in the RHS
-- of the rule. If there is not an object clock in one pattern, then
-- the object is not in the other pattern either
  from
    -- Correspondence between an observer and DSL rules:
    rm : CorrespMM!RuleMatching,
    rDSL : DSLBeh!Rule -- Rule of the DSL
             (rDSL.name=rm.DSLRuleName and not rDSL.lhs.els->
               exists(e|e.oclIsKindOf(DSLBeh!Clock)))
  to
    cLHS : WeaveBeh!Clock(
      id <-'clk',
      pattern <- rDSL.lhs
    ),
    cRHS : WeaveBeh!Clock(
      id <-'clk',
      pattern <- rDSL.rhs
    )
}
----------------------END OF TRANSFORMATION-------------------------
```

C

Metamodels Used Throughout this Dissertation

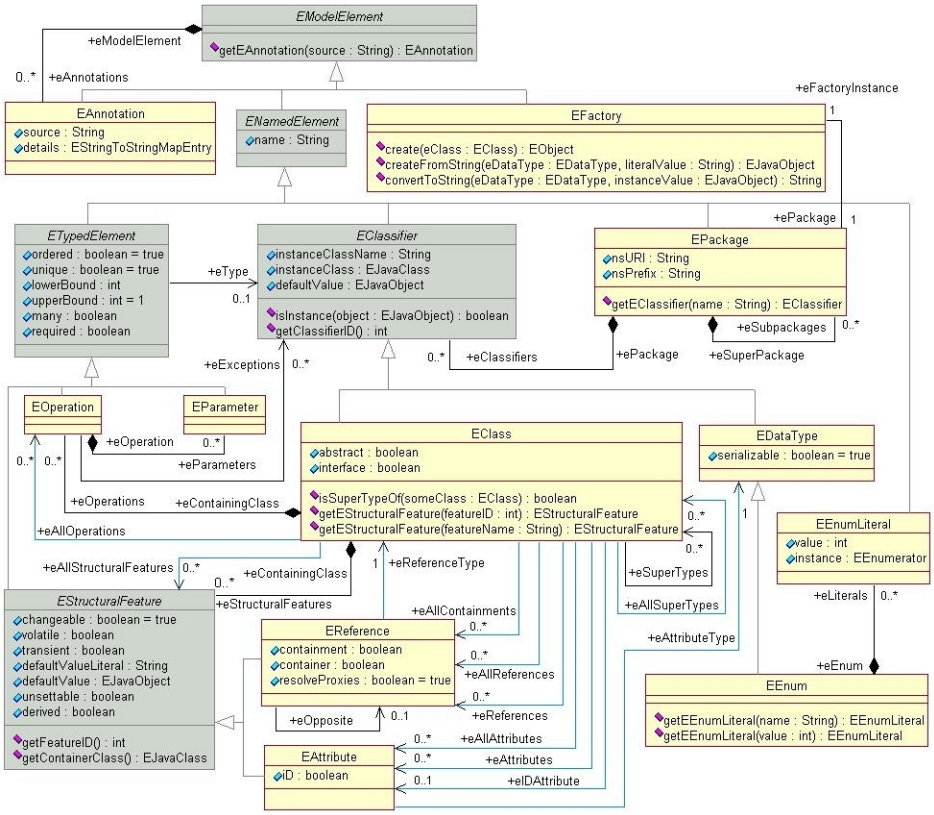


Figure C.1: Ecore metamodel.

This appendix presents the main metamodels used through this dissertation. For example, the *WeaveMetaModel.atl* transformation, whose implementation is presented in Appendix B and which is described in Section 6.2, contains models as input and output whose metamodels are the Ecore and GCS metamodels. In turn, the *WeaveBeh.atl* transformation, presented in the same appendix and section,

contains models as input and output whose metamodels are the Ecore, GCS and Behavior metamodels. Our ePMIF metamodel, in turn, presented and described in Section 5.2, has been developed conforming to the Ecore metamodel.

Ecore, GCS and Behavior metamodels

The Ecore metamodel is shown in Figure C.1. It is the meta-metamodel used by Eclipse to build any metamodel. All the metamodels that appear in this document have been developed conforming to the Ecore metamodel.

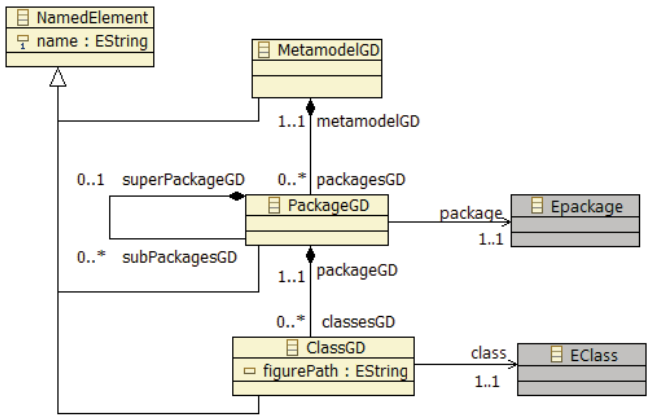


Figure C.2: GCS metamodel.

The GCS metamodel, used to define the concrete syntax of systems in *e-Motions*, is the one shown in Figure C.2. More specifically, it is used to describe the concrete syntax, in visual terms, of a metamodel. For this reason, it defines associations of a visual icon with the non-abstract classes of a metamodel. Hence, it contains references to classes in the Ecore metamodel. The PackageGD class has a reference, package, to the Epackage class, and the ClassGD class points to the EClass class. This means that whenever we are dealing with a GCS model, it has to have an Ecore model associated, which is the metamodel whose concrete syntax is defined in the GCS model.

The Behavior metamodel, to which the sets of in-place behavioral rules for systems modeled in *e-Motions* conform, is presented in Figure C.3. This metamodel is first described when presenting the *e-Motions* tool, in Section 2.2.1.

In a behavioral model containing a set of behavioral rules, the objects present within the patterns of the rules (LSH, RHS and NACs) need to have information of the actual class they represent, and also of their concrete syntax. The same happens with links and slots: they need to know which reference and structural feature they represent. Hence, this metamodel contains references to the Ecore and GCS metamodels. In this way, the Link and Slot classes reference the EReference and EStructuralFeature classes in the Ecore metamodel, respectively. The Behavior and Object classes, in turn, point to the MetamodelGD and ClassGD classes in the GCS metamodel.

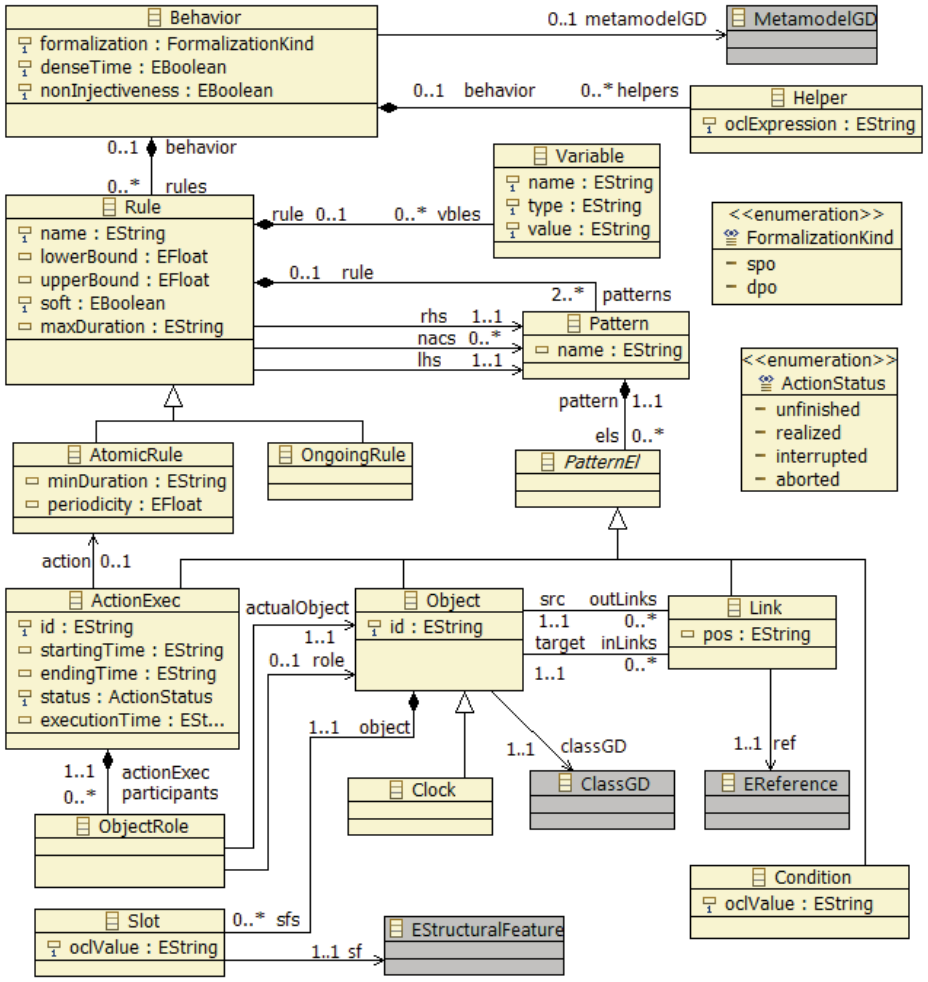


Figure C.3: Behavior metamodel.

Due to this dependency between models described, the *Allow inter-model references* option has to be enabled when defining ATL transformations with models conforming to these metamodels (see for example the ATL transformations presented in Appendix B).

eMotions-ePMIF Metamodel

Chapter 5 presents our approach for defining queuing network models (QNMs) in our xQNM tool. As explained in Section 5.3.2, the QNMs introduced by users in the tool conform the the ePMIF metamodel shown in Figure 5.4. Such models are transformed to an internal representation in our tool which conforms to a slightly different metamodel. The new representation makes the model more compact and efficient, and it is important for conducting the simulations. Both the internal xQNM metamodel and the new representation are transparent to users.

Here, we show such metamodel (let us name it eMotions-ePMIF, Fig. C.4) and presents its differences with regards to the ePMIF metamodel (Figure 5.4). These are the following:

- In ePMIF, transitions are specified for objects Workload (both OpenWorkload and ClosedWorkload) and ServiceRequest. In eMotions-ePMIF, there are no transitions for ClosedWorkload objects because they are not needed. As for OpenWorkload objects, we have made a specialization: OWL1T for OpenWorkloads that always enter the same server and OWLnT for OpenWorkloads when there is more than one server to which transition is possible when the jobs enter the system. The same thing happens with ServiceRequests, where the same specialization has been done. This specialization has been done to improve the efficiency of the behavioral rules simulation, since matchings are performed easily.

- In eMotions-ePMIF, we have only one type of ServiceRequest instead of three. It is TServiceRequest (for "time service request"). It is due to some reasons. Firstly, if we have only one type of object for service requests, then we need fewer behavioral rules to model the system and, consequently, it is more efficient. Secondly, WorkUnitServiceRequest objects can be directly converted to TimeServiceRequest objects as long as WorkUnitServer objects are converted to Server objects. Finally, we can also convert DemandServiceRequest objects to TimeServiceRequest objects because the service time of the TimeServiceRequest can be obtained dividing the service demand of the DemandServiceRequest by the number of visits. Now, being all service requests of type TServiceRequest, we can see that this class has many attributes. The attributes serviceDistr and serviceParams are those used for service times of jobs. Attributes wklds, tS and aS are used for modeling job flow.

- There are no WorkUnitServer objects anymore, they are all of type Server now. This is directly related to the previous point: if there are no WorkUnitServiceRequests, then we do not need WorkUnitServers (and viceversa).

- In ClosedWorkloads, we do not specify parameters for think time. The reason is that the thinkDevice of a ClosedWorkload is nothing but a Server, just as the rest of servers. As a consequence, we have considered that the think time (i.e, its probability distribution and parameters) is specified by a TServiceRequest, just like for the other servers. However, we do have a centralSrv reference to a Server across which all jobs have to flow.

- Attributes transitProbs are not of type Double anymore but of type Integer. They contain the probabilities in a scale from 1 to 10000. They represent the same probabilities as in the ePMIF metamodel, but multiplied

by 10000. Furthermore, in this new attribute the probability in position *n* in the sequence contains the value in the position *n* in the sequence in ePMIF plus the addition of all the previous values (all multiplied by 10000). This representation improves the efficiency of transitions in simulations.

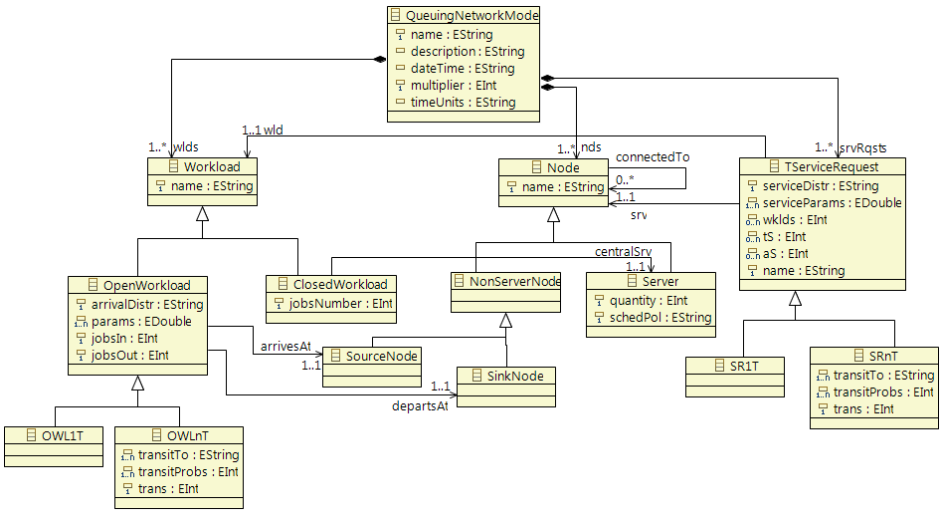


Figure C.4: eMotions-ePMIF metamodel.

D
Resumen

El objetivo de esta tesis es proponer un enfoque dirigido por modelos para analizar propiedades de rendimiento y fiabilidad en sistemas dinámicos. Dicho enfoque ha de ser capaz de analizar estas propiedades en las primeras fases del desarrollo de los sistemas. El uso de técnicas de Ingeniería Dirigida por Modelos (del inglés, *Model-Driven Engineering*) nos permite abstraer y prescindir de detalles de la implementación y centrarnos en el comportamiento del sistema sin tener en cuenta ninguna plataforma o medio concretos de despliegue. En la Sección D.1 motivamos el problema, la Sección D.2 describe nuestro enfoque, la Sección D.3 lo usa para construir un lenguaje visual de dominio específico para representar y simular modelos de redes de colas y la Sección D.4 presenta un enfoque alternativo, más modular que el original.

D.1 Motivación

A la hora de desarrollar cualquier tipo de sistema, lo primero en lo que se piensa es en la funcionalidad del mismo. Se intenta identificar las acciones o funciones que el sistema debe llevar a cabo, mediante la identificación de sus propiedades funcionales. Sin embargo, en muchos sistemas es muy importante no pasar por alto los requisitos no funcionales, que idenfitican *cómo* se debe comportar el sistema.

Pensemos por ejemplo en sistemas implementados total o parcialmente mediante software, como los sistemas críticos. Cuanto más crítico es el software, más importante es que se satisfagan sus requisitos no funcionales. Hay casos conocidos en los que la no satisfacción de las propiedades no funcionales de los sistemas han dado lugar a grandes pérdidas de dinero, cosas materiales, o incluso personas. La tesis de Rodríguez-Dapena [RD02] describe algunos de esos casos. Por mencionar algunos, 130 personas resultaron heridas en 1998 debido al despliegue repentino del airbag en algunos automóviles. General Motors tuvo que retirar casi un millón de vehículos debido a este fallo. El problema era que los vehículos tenían un sensor mal calibrado que hacía que se desplegara el airbag en vías mal pavimentadas o con adoquines. El arreglo consistió en una simple reprogramación del software del sensor. En 1996, un problema en el software

empotrado en los semáforos de Washington D.C. fastidió mucho a las personas que tenían que llegar al trabajo a su hora. La mayoría de los semáforos del centro de la ciudad comenzaron a funcionar como lo hacen los fines de semana, donde la luz verde dura solamente 15 segundos, en vez de funcionar como lo hacen a las horas punta entre semana, donde la luz verde se mantiene durante 60 segundos. Esto causó largos atascos a la hora de entrar a trabajar. El problema se debió a la nueva versión del software instalado en el sistema central que controla todos los semáforos. También fue un problema de software lo que hizo que la sonda espacial *Mars Polar Lander* se estrellase sobre la superficie de Marte en 1999. El software era responsable de encender los motores que harían que la sonda frenase antes de aterrizar, pero estos motores no se encendieron, por lo que la sonda se precipitó sobre la superficie a gran velocidad. Todos los fallos mencionados se podrían haber evitado si los requisitos no funcionales se hubiesen identificado, definido, diseñado, implementado y probados adecuadamente.

Una vez se tiene claro que los requisitos no funcionales deben ser tenidos en cuenta, algo importante es *cuándo* y *cómo* deberían ser especificados, implementados y probados. Muchas propuestas cometen el error de probar los requisitos no funcionales de los sistemas en las últimas etapas del proceso de construcción de los sistemas. La Figura D.1 muestra varios enfoques para la comprobación de los requisitos no funcionales durante el proceso del desarrollo de software. Consideramos que los enfoques a) y b) no se deberían emplear, mientras que el enfoque c) es el ideal. Los tres ejemplos descritos anteriormente siguen el enfoque a). Por ejemplo, el fallo en los semáforos no se detectó hasta que estos estaban en funcionamiento y crearon el caos. También es común aplicar ingeniería inversa no deseada. Idealmente, el diseño define qué se debe implementar. Sin embargo, en muchos proyectos de la vida real, éste no es siempre el caso. De hecho, durante la fase de implementación, los ingenieros a menudo se dan cuenta de que se debe redefinir el diseño, así que deciden modificar primero la implementación y luego propagar los cambios hacia el diseño—enfoque b). También es muy común que los proyectos se retrasen y la fecha de entrega llegue antes de lo esperado. En estos casos, la fase de implementación se olvida completamente de la fase de diseño, y los cambios que se realizan en la implementación ni siquiera son reflejados en el diseño. Esto es especialmente perjudicial en el mantenimiento de los sistemas a largo plazo, pues el diseño y la implementación no estarán sincronizados.

En los sistemas software, las fases de diseño, implementación y despliegue se realizan en el mismo medio: el ordenador. Aplicar cambios desde el despliegue hacia la implementación (enfoque a en la Figura D.1) o desde la implementación hacia el diseño (enfoque b) es perjudicial en estos sistemas. Sin embargo, al menos, el medio no se cambia al aplicar dichos cambios. La situación es más crítica en el caso de sistemas no basados en software, o aquellos en los que el software representa una pequeña parte del sistema, como los sistemas empotrados. En ellos, el diseño y la implementación de prototipos se pueden realizar en un ordenador, pero el despliegue real del sistema se debe hacer en un medio físico

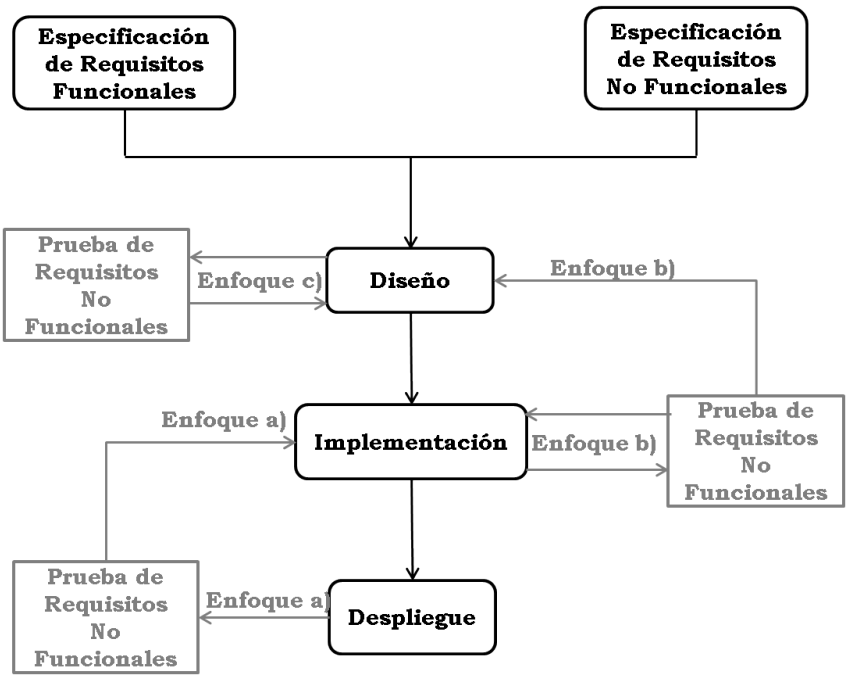


Figure D.1: Propiedades no funcionales en el proceso de desarrollo de software

con materiales reales. Aplicar cambios en estos casos siempre será caro, pues se tendrá que cambiar ciertos componentes del sistema o incluso desecharlos tras ser desplegado el sistema. Los tres ejemplos presentados anteriormente corresponden a este tipo de sistemas.

D.1.1 Enfoque y Metodología

Uno de los objetivos de esta tesis es el de ser capaces de especificar, implementar y probar los requisitos no funcionales antes de la implementación de los sistemas (enfoque c de la Figura D.1) y, por tanto, independientemente de la plataforma. Para ello, hacemos uso de la metodología conocida como Ingeniería Dirigida por Modelos (MDE, del inglés, *Model-Driven Engineering*). MDE nació como una propuesta prometedora para tratar la complejidad de las diferentes plataformas y superar la inhabilidad de los lenguajes de tercera generación para paliar dicha complejidad y expresar los conceptos del dominio de manera efecitva. MDE se basa en el uso de modelos como entidades de primera clase a lo largo de todo el proceso de desarrollo de software. De acuerdo a [KWB03], "Un modelo es una descripción de (parte de) un sistema escrito en un lenguaje bien definido. Un lenguaje bien definido es un lenguaje con una forma (sintaxis) y significado (semántica) bien definidos, que es adecuado para ser interpretado automáticamente por un ordenador". Así, en MDE los modelos no se usan como mera documentación, sino que se convierten en artefactos clave a partir de los cuales se puede

derivar e implementar el sistema completo.

Las tecnologías de MDE aplican lecciones aprendidas de esfuerzos anteriores y ofrecen muchos mecanismos para llevar a cabo abstracciones de la plataforma y emplear lenguajes de alto nivel. En este sentido, en vez de usar lenguages y notaciones de propósito general, como los lenguajes C o Java, que raramente se usan para expresar conceptos del dominio de aplicación o notación relativa al diseño, los lenguajes de dominio específico (DSL, del inglés, *domain-specific language*) se pueden crear mediante metamodelado para ajustarse a la sintaxis y la semántica del dominio, por lo que son más cercanos al dominio del problema que al de aplicación. Contar con lenguages de dominio específico permite asignar elementos gráficos a un dominio conocido. Esto da lugar a los llamados lenguages visuales de dominio específico (DSVL, del inglés, *domain-specific visual language*). No sólo ayudan a reducir el proceso de aprendizaje, sino que también ayudan a un amplio conjunto de expertos en distintas materias, como diseñadores de sistemas y arquitectos software, a asegurarse de que el sistema cumple con las necesidades del usuario. Una tecnología MDE crucial son las transformaciones de modelo, cuya aplicación se ha vuelto fundamental, estando disponibles algunas herramientas y lenguajes de transformación estables.

Además, los generadores de herramientas MDE no tienen que ser tan complicados como los de los ochenta y los noventa, pues pueden sintentizar artefactos que se mapean con APIs de plataformas middleware de alto nivel, en vez de mapearse con APIs de sistemas operativos de bajo nivel. Por tanto, a menudo es más fácil y ligero desarrollar, depurar y evolucionar herramientas y aplicaciones MDE creadas con estos generadores. MDE ya integra muchas tecnologías tales como patrones, model checkers, lenguages orientados a aspectos y de tercera generación, frameworks de aplicación, plataformas middleware basadas en componentes, arquitecturas de líneas de productos, etc.

Actualmente existen varias propuestas para modelar la estructura y el comportamiento funcional de los sistemas en las primeras fases del desarrollo de software, pero no tienen en cuenta los requisitos no funcionales. Algunas de estas propuestas también cuentan con entornos donde animar o ejecutar las especificaciones, basados en la transformación de los modelos a otros que pueden ser ejecutados [dLV06, dLV08, EHC05, EE08, EHKZ05a].

Como ya se ha mencionado, la especificación correcta y completa de un sistema debe incluir los requisitos no funcionales. En los últimos años, los investigadores se han enfrentado al reto de definir modelos cuantitativos para definir y especificar los aspectos no funcionales de los artefactos software [BMIS04, Zsc10]. Se han introducido varias metodologías, todas combinando la idea de anotar los modelos software con información relacionada con aspectos no funcionales, y luego transformar estos modelos anotados a modelos capaces de ser validados [CMI07, FBV$^+$09, FJA$^+$09]. La mayoría de estas propuestas para anotar modelos con requisitos no funcionales existen para notaciones basadas en UML, con Perfiles UML tales como UML-QoSFT, UML-SPT o

Marte [OMG05, OMG04, OMG08]. Hoy día existen varias herramientas para analizar y simular estos modelos y para llevar a cabo análisis de rendimiento.

A pesar de que estos perfiles proporcionan soluciones para UML, la situación es diferente cuando se utilizan DSL (o DSVL) para especificar un sistema. Para empezar, los requisitos no funcionales se suelen escribir usando lenguajes totalmente extranjeros para los diseñadores del sistema, pues dichos lenguajes se ven influenciados por los métodos de análisis a usar, y son escritos en los lenguajes de estos métodos. Además, su nivel de abstracción suele ser menor que el de las notaciones DSL que se usan para especificar el sistema, y tienden a ser más cercanos al dominio de solución que al dominio del problema (en contraste con la orientación problema-dominio de los DSL). Además, cuando se tienen que analizar varias propiedades, los modelos anotados se ven inundados de marcas y anotaciones diferentes (como la mayoría de los diagramas que aparecen en la especificación de MARTE [OMG08]). Otro problema es que las propuestas actuales para la especificación de estas propiedades requieren tener conocimiento sobre lenguajes y notaciones especializados, que choca con la naturaleza intuitiva de los DSL y dificulta una combinación ligera y directa con estos lenguajes. Finalmente, la mayoría de las propuestas especifican requisitos no funcionales y restricciones usando un enfoque *preceptivo*, es decir, anotan los modelos con un conjuto de requisitos sobre el comportamiento del sistema y con restricciones sobre ciertos elementos del modelo, pero no son muy expresivos para describir cómo estos valores son computados dinámicamente o evolucionan en el tiempo.

D.1.2 Contribución

En esta tesis presentamos una propuesta innovadora para especificar las propiedades no funcionales que han de ser analizadas y probadas, integrando nuevos objetos en las especificaciones que permiten caputar dichas propiedades en las primeras fases del desarrolo de software—enfoque c) en la Figura D.1. De entre los requisitos no funcionales existentes, como seguridad, usabilidad, fiabilidad o rendimiento, aquí nos centramos en los dos últimos. Nuestra propuesta se basa en la *observación* de la ejecución de las acciones del sistema y del estado de los objetos que lo constituyen, y la empleamos para el análisis y la simulación de lenguajes visuales de dominio específico (DSVL) que especifican el comportamiento del sistema mediante reglas que describen cómo evolucionan los artefactos modelados a través del tiempo. Así, dada una especificación inicial del sistema, el uso de objetos *observadores* permite analizar requisitos no funcionales en las primeras fases del ciclo de desarrollo del sistema. La clave de estos observadores está en que se pueden definir y especificar en el mismo lenguaje que el experto del dominio usa para describir el sistema, por lo que no se necesita ninguna notación adicional. Así, la idea es mezclar la especificación del sistema y de los observadores, combinándolos desde la fase de diseño (Figura D.2), de modo que las propiedades no funcionales queden especificadas en los observadores.

No solamente consideramos los sistemas software, sino también aquellos que

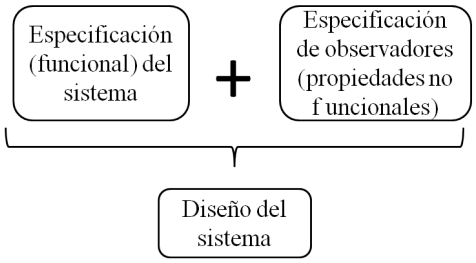


Figure D.2: Integrando observadores desde la especificación.

son desplegados en un medio diferente al del ordenador. Concretamente, nos centramos en *sistemas dinámicos*. Por "dinámicos", entendemos aquellos sistemas en los que hay *cosas* fluyendo. Estas "cosas" pueden ser datos, componentes, partes de componentes, etc. En cuanto a "fluyendo", entendemos que las cosas evolucionan dentro del sistema a lo largo del tiempo, por ejemplo cambiando su estado o moviéndose entre diferentes componentes. A lo largo de esta tesis se presentan varios casos de estudio de sistemas dinámicos. Por ejemplo, un *sistema de línea de producción* de martillos, donde las piezas de los martillos se van moviendo entre las distintas máquinas de la fábrica. Otro ejemplo de sistema dinámico presentado en este documento es un *proceso de facturación* en un aeropuerto, donde los pasajeros atraviesan diferentes estados: esperando en cola, facturación realizada, esperando al avión. etc. Cualquier red de datos también se considera un sistema dinámico, pues los paquetes fluyen a través de los servidores. En esta tesis también tratamos con un formalismo que permite modelar sistemas dinámicos: las redes de colas.

Hemos utilizado una herramienta presentada en una tesis anterior [Riv10] como entorno para nuestra propuesta. Se llama *e-Motions* y permite especificar sistemas basados en tiempo real [RDV09, RVD09, RDV10] mediante un conjunto de reglas de comportamiento. En este documento se llevan a cabo varias extensiones de dicha herramienta. También se describe una metodología para la especificación de propiedades no funcionales en DSVL basados en reglas, y se presenta un enfoque modular para desacoplar la especificación de observadores con respecto a la especificación funcional del sistema.

En esta tesis también nos enfrentamos al reto de la comunidad MDE de construir puentes semánticos entre diferentes espacios de modelado [DGFD06], de modo que los programas o modelos puedan viajar desde un espacio a otro, pudiendo aprovechar diferentes herramientas de análisis para razonar sobre las propiedades del sistema, llevar a cabo simulaciones, validaciones, etc. [Val08]. Estos puentes se pueden especificar en el contexto MDE de manera natural mediante transformaciones de modelo. Establecer puentes entre diferentes dominios es más sencillo cuando estos tienen una expresividad similar, ya que en este caso las transformaciones de modelo se pueden definir de un modo directo. Sin embargo, la situación es más complicada cuando la expresividad de los dos dominios a

relacionar es muy dispar. En nuestro caso, en un principio estábamos interesandos en mapear nuestra notación y herramientra para modelar sistemas en tiempo real, *e-Motions*, con otras notaciones y herramientras a más bajo nivel de abstracción (Figura D.3). El objetivo era el de llevar a cabo diferentes tipos de análisis sobre los modelos definidos en *e-Motions*, aparte de los permitidos en dicha herramienta. Una de nuestras primeras elecciones fue las redes de colas, puesto que permiten realizar análisis del rendimiento sobre los modelos de modo analítico [DB78]. Sin embargo, pronto nos dimos cuenta de que la mayoría de la información sobre los modelos descrita en *e-Motions* se perdía o degradaba al trasladar los modelos a redes de colas. Por ejemplo, las expresiones OCL, muchas veces bastante complejas, con algoritmos para seleccionar dónde mover los objetos en algunos sistemas, se reducían simplemente a probabilidades, que tenían que ser estimadas de algún modo (normalmente mediante simulación, que era precisamente lo que queríamos evitar al usar métodos analíticos). Del mismo modo, no era una tarea trivial identificar los elementos del modelo en *e-Motions* que deberían ser transformados a servidores, trabajos o colas; es más, esto no era posible sin intervención humana, algo que se busca reducir al mínimo en la construcción de puentes semánticos.

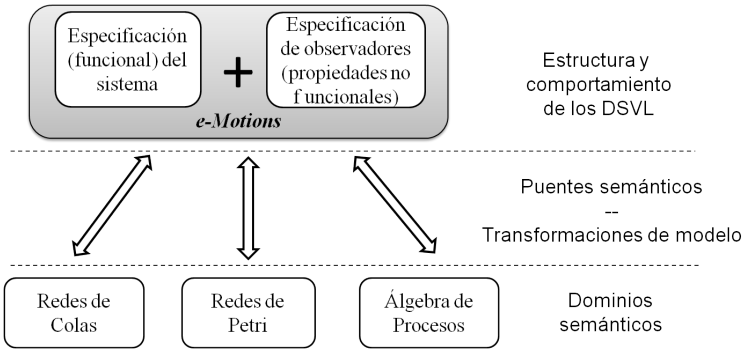


Figure D.3: Puentes semánticos.

Sin embargo, nos dimos cuenta de que el mapeo hacia el otro lado no sólo era posible, sino sencillo de implementar. El construir un puente entre redes de colas y *e-Motions* nos permitiría usar las herramientras de simulación y análisis de *e-Motions* con los modelos de las redes de colas, lo cual proporciona algunos beneficios destacables. Por ejemplo, se podría utilizar cualquier distribución de probabilidad para los tiempos de llegada y servicio de trabajos, aparte de las habituales distribuciones exponencial y uniforme; también se podría hacer uso de las ventajas de MDE para construir, de un modo modular y a alto nivel, un lenguaje visual de dominio específico para modelar redes de colas y tener un kit de herramientas para simularlos; podríamos usar (una versión Ecore de) PMIF [Smi10], un formato de intercambio de modelos de rendimiento, para definir e intercambiar modelos de redes de colas entre distintas herramientas; finalmente, el puente también permitiría definir una semántica de comportamiento para las redes de colas,

especificada mediante un conjunto fijo de reglas de comportamiento especificadas en *e-Motions*. En cuanto a este último punto y teniendo en cuenta el poder expresivo y las posibilidades de modelado de DSVL en *e-Motions*, se podrían decorar las reglas semánticas con nuevas características con el objetivo de extender el comportamiento de las redes de colas de un modo más real. Concretamente, hemos extendido dicho comportamiento considerando que los servidores pueden fallar o caerse, y estar no disponibles durante un cierto tiempo. Esta tesis expone cómo se construyó este puente semántico, y describe la herramienta (llamada xQNM) y el DSVL que hemos desarrollado usando técnicas MDE estándares. Además, comparamos xQNM con otras herramientas de redes de colas que fueron desarrolladas usando enfoques de programación estándar, a pesar de que nuestra herramienta está definida a un mayor nivel de abstracción. Presentamos la arquitectura de xQNM, basada en niveles (cada uno a un nivel de abstracción diferente) y los detalles de implementación que contribuyen a conseguir las ventajas mencionadas anteriormente.

D.2 Análisis Dirigido por Modelos de DSVL Basados en Reglas

En esta sección resumimos nuestro enfoque para analizar y probar propiedades no funcionales en lenguajes visuales de dominio específico (DSVL) basados en reglas en las primeras fases del desarrollo de los sistemas.

D.2.1 Modelado Funcional de Sistemas

En nuestra propuesta, expresamos el comportamiento dinámico de los sistemas mediante lenguajes visuales de dominio específico (DSVL) que describen la evolución de los artefactos modelados a lo largo del tiempo. En MDE, esto se puede hacer utilizando transformaciones de modelo que soporten actualizaciones *in-place* [CH03]. Así, el comportamiento del sistema se especifica a través de las acciones permitidas, que a su vez se modelan con reglas de transformación de modelos.

A continuación presentamos un ejemplo de una línea de producción de martillos que, como cualquier DSVL, se define en términos de tres elementos principales: la sintaxis abstracta, la sintaxis concreta y la semántica. La sintaxis abstracta define los conceptos del dominio que el lenguaje es capaz de representar, y se especifica mediante un metamodelo. La sintaxis concreta define la notación del lenguaje, y en nuestro caso la definimos asignando un icono gráfico a cada concepto del metamodelo. La semántica describe el significado de los modelos representados en el lenguaje, y en el caso de modelos de sistemas dinámicos, la semántica de un modelo describe las consecuencias de ejecutar dicho modelo. En nuestro caso, la semántica la especificamos mediante un conjunto de reglas de comportamiento.

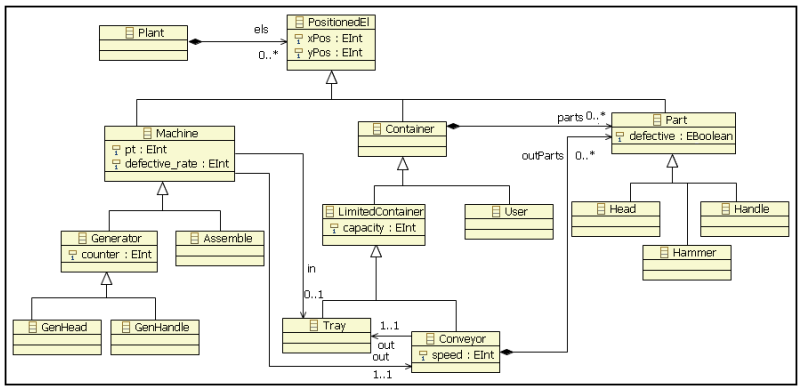


Figure D.4: Metamodelo de la Línea de Producción.

En la Figura D.4 aparece el metamodelo de nuestro sistema. Hay diferentes tipos de máquinas (generadores de mangos, generadores de cabezas y ensambladores), contenedores (usuarios y contenedores con capacidad limitada: bandejas y cintas transportadoras) y piezas (mangos, cabezas y martillos). Todas tienen una posición en la fábrica, indicada por dos coordenadas. Los generadores producirán tantas piezas como su contador indica, y las depositan en cintas transportadoras, las cuales mueven piezas desde las máquinas hasta las bandejas a una velocidad determinada; los ensambladores cogen piezas que están en bandejas para crear martillos, los cuales depositan en cintas transportadoras y son finalmente recolectados por usuarios. Las piezas pueden ser o no defectuosas. Las máquinas trabajan de acuerdo a un tiempo de producción (pt) que indica el número medio de unidades de tiempo que tardan en producir una pieza. Tienen otro atributo, defective_rate, que determina el porcentaje de piezas defectuosas que producen y que está directamente relacionado con el tiempo de producción (normalmente, cuanto más rápido generen las piezas, más piezas defectuosas producirán). Las bandejas y las cintas transportadoras pueden contener el número de piezas que sea, siempre que no supere el valor dado por su atributo capacity.

La Figura D.5 muestra un modelo del sistema, realizado en *e-Motions*, donde aparece un conjunto de objectos y de valores para sus atributos. Este modelo se usa como configuración inicial de nuestro sistema. En la figura se puede observar la sintaxis concreta dada a los conceptos del metamodelo.

A continuación, el comportamiento funcional del sistema se especifica mediante un conjunto de reglas, donde cada una de ellas representa una acción posible. Para nuestro sistema, dicho comportamiento es el que se ha descrito anteriormente con palabras, y ahora hay que especificarlo en *e-Motions*. En nuestro ejemplo, el comportamiento se describe con 6 reglas: una para generar mangos (GenHandle), una para generar cabezas (GenHead), una para describir cómo las cintas transportadores mueve piezas (Carry); una para ensamblar cabezas y mangos y producir así martillos (Assemble), una para depositar piezas en bandejas una vez han

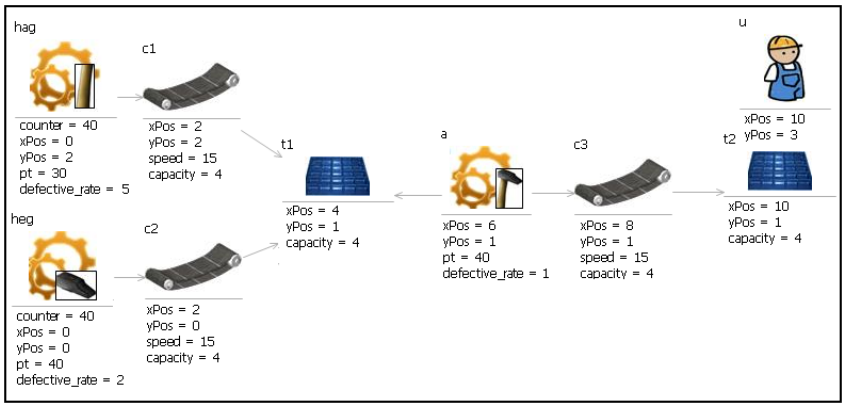


Figure D.5: Modelo del sistema, a ser usado como configuración inicial.

sido transportadas a través de cintas transportadoras (Transfer), y una regla para describir cómo un usuario recolecta un martillo ensamblado (Collect).

Por ejemplo, la Figura D.6 muestra la regla atómica Assemble, que especifica cómo se ensambla un martillo a partir de un mango y una cabeza. En la parte izquierda de la regla, el mango y la cabeza están contenidos en la bandeja de entrada de un asemblador, y la restricción OCL indica que debe haber espacio disponible en la cinta transportadora de salida. En la parte derecha de la regla, la cabeza y el mango han sido consumidos y se ha creado un martillo que se ha depositado sobre la cinta.

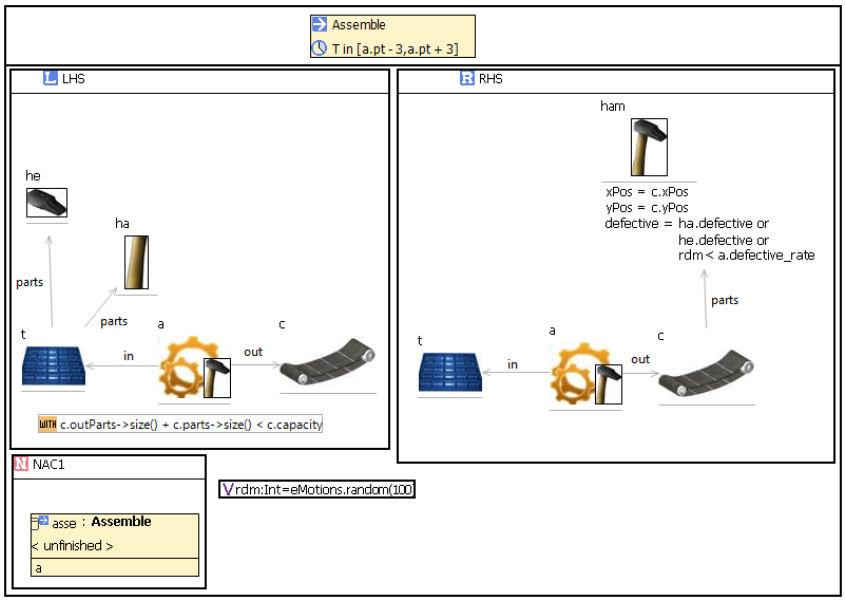


Figure D.6: Regla Assemble.

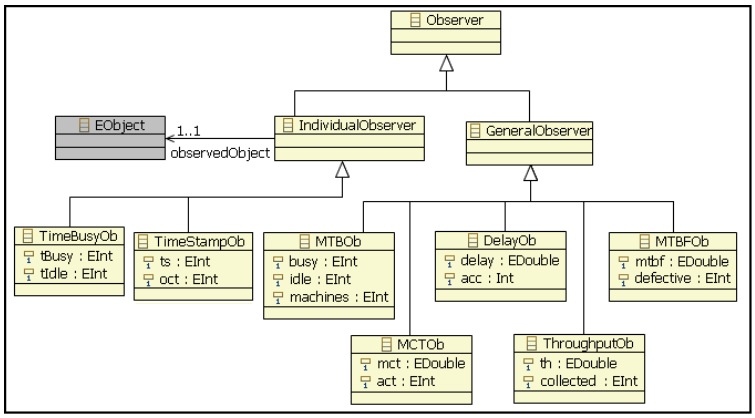


Figure D.7: Metamodelo de Observadores.

D.2.2 Monitorización del Sistema con Observadores

Una vez contamos con lenguajes que nos permiten especificar modelos y su comportamiento, el siguiente paso es el de desarrollar un mecanismo para poder analizar las propiedades no funcionales de los sistemas. Para ello, primero hay que identificar las propiedades a analizar (como retrasos, tiempos de ciclo o tiempos de espera), y además necesitamos un motor de simulación que ejecute las especificaciones. En este resumen nos vamos a centrar en el tiempo de trabajo de las máquinas. Así, para cada máquina, queremos saber el tiempo que ha estado trabajando, así como el tiempo medio de trabajo de todas las máquinas.

En nuestro enfoque contamos con observadores *individuales* y observadores *generales*. Los primeros se utilizan para monitorizar objetos concretos, mientras que los segundos se utilizan para calcular valores agregados, y normalmente utilizan los valores de observadores individuales. La estructura de los observadores se define con un metamodelo, al igual que se define la estructura del sistema. En nuestro caso, este metamodelo es el de la Figura D.7. En este resumen sólo nos centramos en el observador individual TimeBusyOb y en el observador general MTBOb (del inglés, *Mean Time Busy*).

En *e-Motions*, el metamodelo del sistema y el metamodelo de los observadores se combinan, de modo que los observadores individuales pueden apuntar a objetos concretos del sistema y todos los observadores pueden incluirse en las reglas de comportamiento. Así, para registrar el tiempo de trabajo de los ensambladores, en la regla Assemble mostrada anteriormente incluiremos un observador individual de tipo TimeBusyOb, tal y como se muestra en la Figura D.8. En la regla, el observador individual aparece asociado al ensamblador. En la parte derecha de la regla, el valor del atributo tBusy del observador se incrementa con el tiempo que ha tardado la regla en ejecutarse, que es precisamente el tiempo que el ensamblador ha estado ocupado ensamblando el martillo. Se observa así que no se necesita ninguna notación adicional para actualizar el valor del observador.

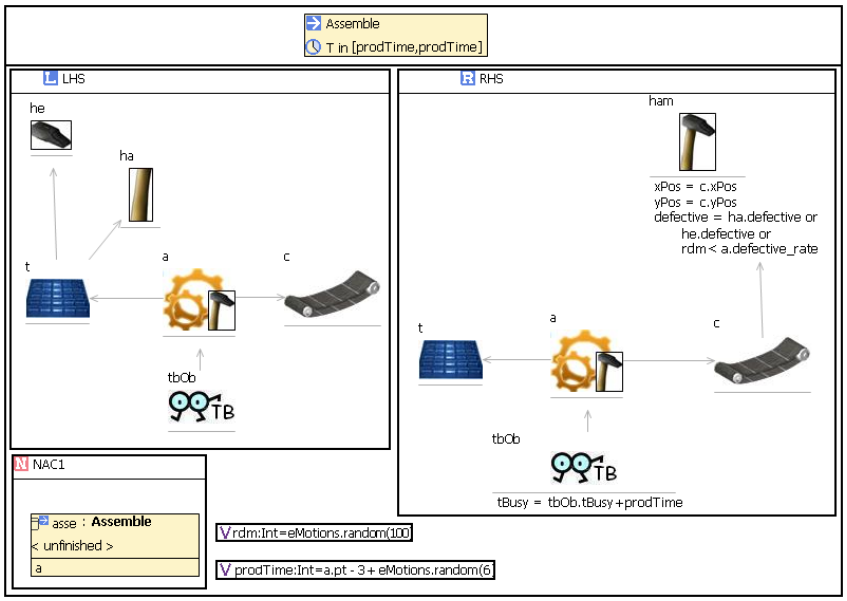


Figure D.8: Regla Assemble con observador.

Para actualizar el valor de nuestro observador general MTBOb, vamos a utilizar una regla *ongoing*. Este tipo de reglas se ejecutan mientras el sistema esté activo, por lo que así se consigue mantener los valores monitorizados por nuestro observador actualizados. Nuestra regla, llamada UpdateMTBOb, se muestra en la Figura D.9. En la regla simplemente aparece el observador general a actualizar. En la parte derecha, observamos que se ha utilizado una expresión OCL para actualizar el valor del atributo busy. Lo que hace es sumar los tiempos que todas las máquinas han estado ocupadas y divide este valor por el número de máquinas que hay en el sistema.

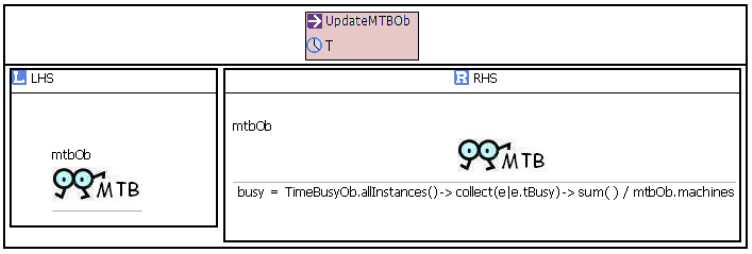


Figure D.9: Regla UpdateMTBOb.

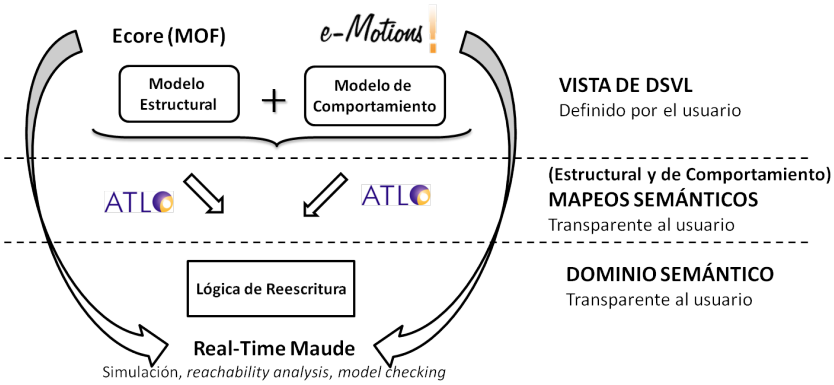


Figure D.10: Mapeo semántico entre *e-Motions* y Maude.

D.2.3 Obtención de los Parámetros No Funcionales del Sistema

Una vez que se ha modelado el comportamiento funcional del sistema y los observadores se han añadido en las reglas, podemos simular nuestras especificaciones y obtener medidas de los parámetros no funcionales del sistema. Para poder realizar la simulación, en *e-Motions* la semántica de las especificaciones se transforma a otro dominio con semántica bien definida, llamado *Real-Time Maude* [ÖM07]. Así, el entorno *e-Motions* no sólo proporciona un editor para describir las especificaciones visuales, sino que también implementa la transformación automática de éstas (usando el lenguaje de transformación de modelos ATL [JABK08b]) a su representación formal en Maude—de forma transparente al usuario. Lo que se obtiene al traducir las reglas a Maude es una especificación del sistema basada en lógica de reescritura. Lo más importante de este enfoque es que permite usar las facilidades y herramientas que Maude ofrece para ejectuar y analizar las especificaciones una vez están expresadas en Maude. Los trabajos [RVD09, RDV10] presentan algunos ejemplos de análisis que se pueden llevar a cabo sobre DSL basados en reglas especificados en Maude. Como hemos dicho, las especificaciones Maude basadas en lógica de reescritura son ejecutables, por lo que se pueden usar como un prototipo del sistema sobre el que realizar distintos tipos de comprobaciones y llevar a cabo simulaciones.

En Maude, el resultado de una simulación es la configuración final de objectos obtenidos tras completar los pasos de reescritura, que no es otra cosa que un modelo. Este modelo devuelto por la simulación se puede transformar de vuelta a un modelo conforme a Ecore, el cual es el mostrado al usuario. Ambas transformaciones, de *e-Motions* a Maude y viceversa, son completamente transparentes al usuario, por lo que éste/a sólo ha de interactuar con el entorno visual de *e-Motions*, sin necesidad de entender cualquier otro formalismo e ignorando que la simulación la realiza el motor de reescritura de Maude (Figura D.10).

Tras simular un sistema, hay que consultar el valor de los atributos de los observadores para saber cómo el sistema se ha comportado. La Tabla D.1 muestra

Table D.1: Resultados de los observadores tras una simulación con un ensamblador (pt = 40 y defective_rate = 1), un generador de mangos (pt = 30 y defective_rate = 5) y un generador de cabezas (pt = 40 y defective_rate = 2).

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.39	mtbf:9'05"	mct:4'05"	delay:2'26"	tBusy:71%	tBusy:95.4%	tBusy:95.4%
collected:38	defective:2	act:163'15"	acc:97'10"	tIdle:29%	tIdle:4.6%	tIdle:4.6%

Table D.2: Resultados de los observadores tras una simulación con un ensamblador (pt = 40 and defective_rate = 1), un generador de mangos (pt = 30 and defective_rate = 5) and un generador de cabezas (pt = 30 and defective_rate = 5).

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.41	mtbf:8'59"	mct:3'42"	delay:2'04"	tBusy:72.4%	tBusy:73.6%	tBusy:96.3%
collected:38	defective:2	act:148'3"	acc:82'28"	tIdle:27.6%	tIdle:26.4%	tIdle:3.7%

los resultados de la simulación para el modelo inicial mostrado en la Figura D.5. En la tabla se muestra el valor de *throughput*, que mide el número de martillos producidos por unidad de tiempo; el tiempo medio de fallos (*MTBF*), con el que modelamos el tiempo medio que transcurre entre la generación de dos martillos defectuosos; el tiempo de ciclo medio (*MCT*), que es lo que tarda en producirse un martillo de media; el retraso (*Delay*), con el que medimos el retraso medio de los martillos con respecto a un tiempo óptimo; y el porcentaje de tiempo que las tres máquinas de nuestro sistema están ocupadas y en espera de trabajo (*HandleGen* es el generador de mangos, *HandleHead* el de cabezas y *Assembler* es el ensamblador).

Tras simular las especificaciones con los parámetros del modelo inicial de la Figura D.5 y estudiar los parámetros no funcionales del sistema, el modelador puede decidir cambiar determinados parámetros del sistema para que el comportamiento se acerque más al deseado en términos de requisitos no funcionales. Así por ejemplo, si lo que se quiere es que el tiempo de ocupación de los dos generadores sea similar, habría que igualar sus tiempos de producción, con lo que se obtendrían los parámetros no funcionales mostrados en la Tabla D.2.

Autoadaptación del Sistema

Además de modificar manualmente el modelo inicial del sistema hasta cumplir con los requisitos no funcionales perseguidos, también se puede añadir reglas para que el sistema se autoadapte automáticamente hasta conseguir el comportamiento deseado. Así por ejemplo, puesto que el tiempo de ciclo medio de los martillos depende de la velocidad de producción de las máquinas (generadores de piezas y ensambladores), se puede aumentar o disminuir dicha velocidad para adaptar el tiempo de ciclo medio.

(a) DecreasePt Rule (b) IncreasePt Rule

Figure D.11: Reglas para modelar la autoadaptación del sistema.

Table D.3: Resultados de los observadores en una simulación donde el sistema se autoadapta con el objetivo de obtener un tiempo de ciclo medio de 1'50".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.7	mtbf:11'30"	mct:1'50.3"	delay:15"	tBusy:95%	tBusy:94.9%	tBusy:93.5%
collected:39	defective:1	act:73'33"	acc:10'10"	tIdle:5%	tIdle:5.1%	tIdle:6.5%

En la Figura D.11 podemos ver dos reglas con las que se pretende que el tiempo de ciclo medio (*MCT*) de los martillos se mantenga alrededor de 110 unidades de tiempo ($= 1'50''$). La primera regla aumenta la velocidad de las tres máquinas del sistema cuando el tiempo de ciclo medio supera 110, mientras que la segunda regla reduce dichos tiempos cuando el tiempo de ciclo medio está por debajo de 110. En la Tabla D.3 se observa que, efectivamente, el tiempo medio de ciclo es muy cercano al valor $1'50''$. Los observadores que monitorizan las demás propiedades no funcionales han computado los valores que aparecen en la tabla. En general, esta simulación ha ofrecido mejores resultados que las dos anteriores, puesto que la velocidad de las máquinas se ha adaptado en tiempo dinámico de acuerdo a las necesidades.

D.3 Especificación y Simulación de Redes de Colas Usando un DSL

Esta sección resume la construcción de un lenguaje de dominio específico (DSL) con el que ser capaces de dibujar, exportar, importar y simular modelos de redes de colas (QNM, del inglés, *Queuing Network Model*). La creación de dicho

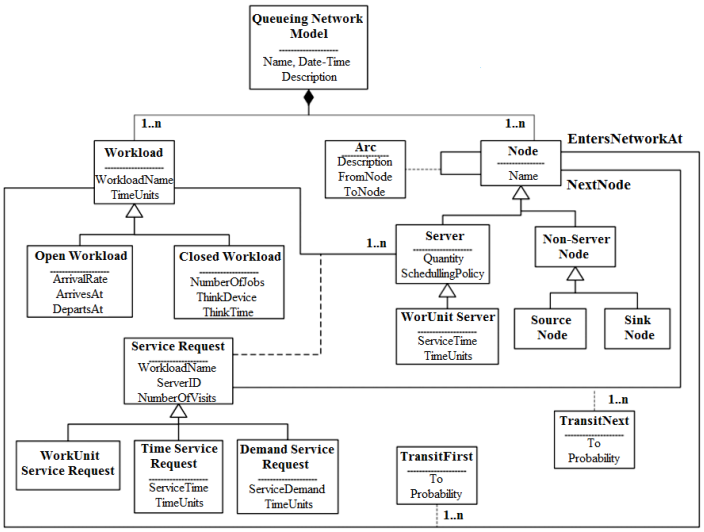


Figure D.12: Metamodelo de PMIF 2 (extraído de [Smi10])

DSL consiste en la construcción de un puente semántico entre el formalismo que analiza las redes de colas y nuestra herramienta *e-Motions*. Así, hemos sido capaces de especificar y modelar el comportamiento de las redes de colas con reglas de comportamiento. Además, se ha creado una herramienta, que es el DSL con el que el usuario interactúa, llamada xQNM construida sobre *e-Motions*. En las siguientes secciones resumimos el DSL creado.

D.3.1 Representación de PMIF en Ecore

PMIF (del inglés, *performance interchange format*) se creó como una representación común para modelar el rendimiento de los sistemas, de modo que los modelos PMIF se pudieran intercambiar entre distintas herramientas que soportasen dicho formato [SW99]. PMIF representa plataformas de computación y redes de interconexión mediante redes de colas y servidores. La primera versión de PMIF se creó en 1998, mientras que la segunda versión, PMIF 2, data de 2004 [Smi04, SL04, Smi10]. El metamodelo para PMIF 2 se muestra en la Figura D.12.

Aunque PMIF se concibió como un formato de intercambio de modelos de redes de colas entre distintas herramientas, son pocas las que hoy día soportan este formato. A veces, se utilizan servicios web que actúan como puentes entre los formatos de varias herramientras. Así, el servicio web coge un modelo de redes de colas expresado en un determinado formato, lo traslada a PMIF, y lo vuelve a transformar en otro formato capaz de ser leído por otra herramienta. Un ejemplo aparece en [RLPS05], donde se utiliza PMIF y un servicio web para intercambiar modelos entre las notaciones de SPE·ED y QNAP.

En esta tesis, proponemos un metamodelo para PMIF conforme a Ecore, de

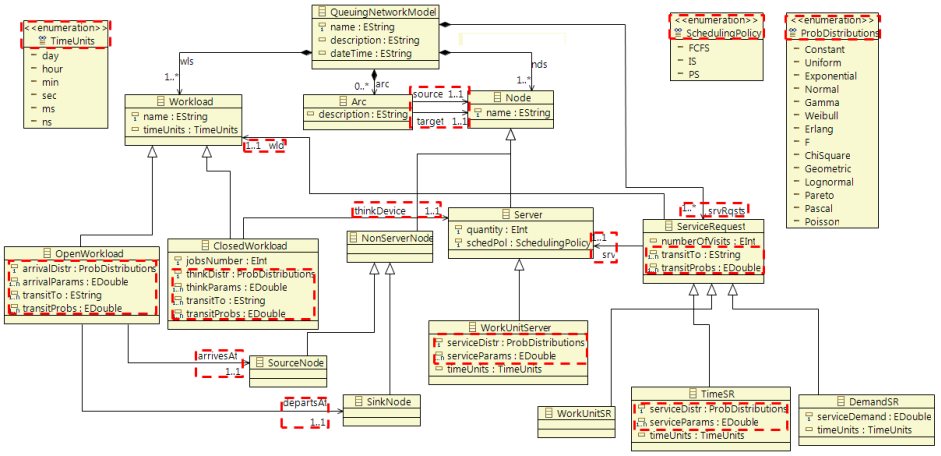


Figure D.13: Metamodelo de ePMIF (conforme a Ecore)

modo que los modelos de redes de colas puedan ser representados en Eclipse e integrados en procesos MDE. Así, hemos utilizado Ecore como meta-metamodelo para definir nuestra versión de PMIF, llamada ePMIF (de Eclipse-PMIF). Nuestro metamodelo para PMIF, ePMIF, se muestra en la Figura D.13. Puede verse como la versión MDE de PMIF 2, con algunos cambios menores.

A continuación describimos algunas de las diferencias entre PMIF 2 y ePMIF, marcadas con cajas rojas en la Figura D.13. En PMIF 2, las probabilidades se expresan como clases en el metamodelo, mientras que en nuestro enfoque se especifican con atributos (para reducir así el número de elementos en los modelos resultantes, por razones de simulación). En nuestro caso, ServiceRequest ya no es una clase de asociación, y también hemos unificado el modo de especificar transiciones en las clases Workload y en ServiceRequest (lo cual es muy útil para especificar el comportamiento). Así, en PMIF 2 se necesitaba un objeto de tipo Transit para cada camino en una división, mientras que en nuestro caso sólo se necesitan dos attributos, independientemente del número de caminos que haya, transitTo y transitProbs. El primero contiene una secuencia con los nombres de los Nodes donde el Workload puede transitar, mientras que el segundo contiene una secuencia con las probabilidades para estas transiciones. Otra diferencia es la forma de especificar los tiempos de llegada y de servicio. En PMIF 2, se especifican con los atributos ArrivalRate y ServiceTime, respectivamente. PMIF 2 asume que estos valores representan el parámetro de distributiones de poisson y exponencial, respectivamente. En nuestro caso, queremos permitir el uso de muchas distribuciones de probabilidad para estos tiempos, por lo que hemos definido un tipo enumerado, ProbDistributions, con las distribuciones que se pueden usar. Por último, en ePMIF usamos referencias en vez de atributos para referirnos a otros objetos. Esto tiene la ventaja de que las referencias no se pueden especificar incorrectamente, mientras que es muy común escribir el nombre de un

objeto de forma errónea. Además, si se cambia el nombre de un objeto el cual es referenciado desde otro mediante su nombre, se producirá una inconsistencia.

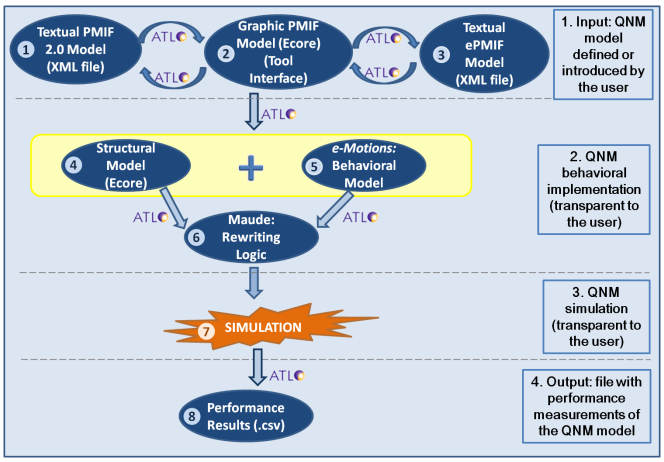


Figure D.14: Arquitectura de xQNM

D.3.2 Visión General de xQNM

A la herramienta que hemos construido para modelar y simular modelos de redes de colas la hemos llamado xQNM (del inglés, *executable Queuing Network Model*). Su arquitectura se muestra en la Figura D.14. La dirección de las flechas indica el proceso de obtención de modelos de rendimiento para los modelos de redes de colas dibujados o importados en la herramienta.

Los óvalos 1, 2 y 3 se corresponden con la interfaz gráfica de nuestra herramienta, donde se pueden dibujar, importar y exportar modelos de redes de colas. En la Figura D.15 se observa dicha interfaz, donde aparece dibujado un modelo con tres servidores, tres *service requests* (uno para cada servidor), un *workload*, un nodo fuente y un nodo sumidero. En el panel de la derecha se seleccionan los elementos que se quieren introducir en el modelo, y en el panel de abajo aparecen las características del elemento seleccionado (un *service request* en la figura). En la herramienta se pueden definir modelos de redes de colas abiertas, cerradas y mixtas, aunque las últimas no se pueden simular en la versión actual de la herramienta.

Al exportar un modelo, esto se realiza hacia un fichero XML que contiene su representación en ePMIF. Además, la herramienta permite exportar también a XML conforme a PMIF 2. Con esto conseguimos una mayor interoperabilidad de nuestra herramienta con las herramientas existentes. A la hora de importar modelos, nuestra herramienta permite la importación de modelos conforme a PMIF 2. Así, se puede importar un modelo para ser simulado en nuestra herramienta. También se puede importar un modelo, modificar el mismo con la interfaz gráfica

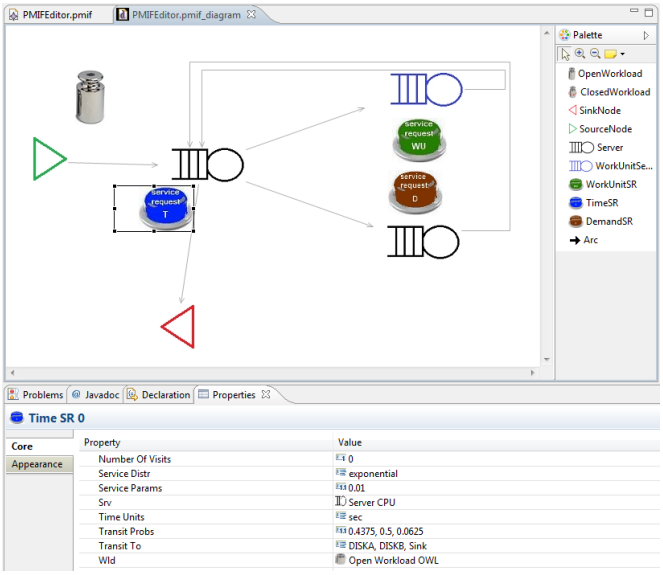


Figure D.15: Editor gráfico de xQNM

de la herramietna, y posteriormente simularlo o exportarlo de nuevo. Por supuesto, también es posible importar modelos conforme a ePMIF.

El óvalo 4 de la Figura D.14 representa la estructura de nuestros modelos (ePMIF más observadores), y el óvalo 5 representa el modelo de comportamiento para las redes de colas definido en *e-Motions*. Los observadores utilizados para monitorizar los parámetros de rendimiento de los modelos son los del metamodelo de la Figura D.16. Hay tres tipos de observadores para monitorizar propiedades de rendimiento de los modelos: WorkloadOb, ServerOb y ServiceRequestOb, que monitorizan las propiedades de los *workloads*, servidores y *service requests*, respectivamente. El observador SimOb contiene los parámetros de la simulación introducidos por el usuario.

Los observadores presentados son utilizados a la hora de definir las reglas de comportamiento (óvalo 5). Básicamente, la especificación del comportamiento cuenta con una regla que modela la llegada de trabajos al sistema (EnterOpenWLFnT) otra para especificar cómo los trabajos transitan entre servidores (TransitJobsnT) y una tercera para modelar la salida de trabajos del sistema (ExitOpenWLF). Por razones de eficiencia, hay variaciones de estas reglas para el caso en que sólo hay un servidor al que los trabajos pueden transitar. Con objeto de mostrar algunas reglas, la EnterOpenWLFnT y la ExitOpenWLF se muestran en la Figura D.17.

Cuando el usuario introduce un modelo de redes de colas en la herramienta xQNM, bien dibujándolo o importándolo, y decide simularlo, el modelo es automáticamente transformado a su representación en *e-Motions* (transformación desde el óvalo 2 a los óvalos 4 y 5 de la Figura D.14). Posteriormente, estos

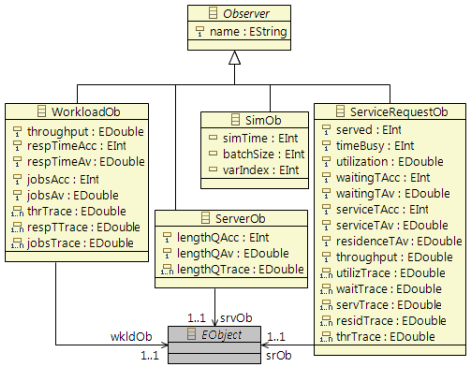


Figure D.16: Metamodelo de Observadores

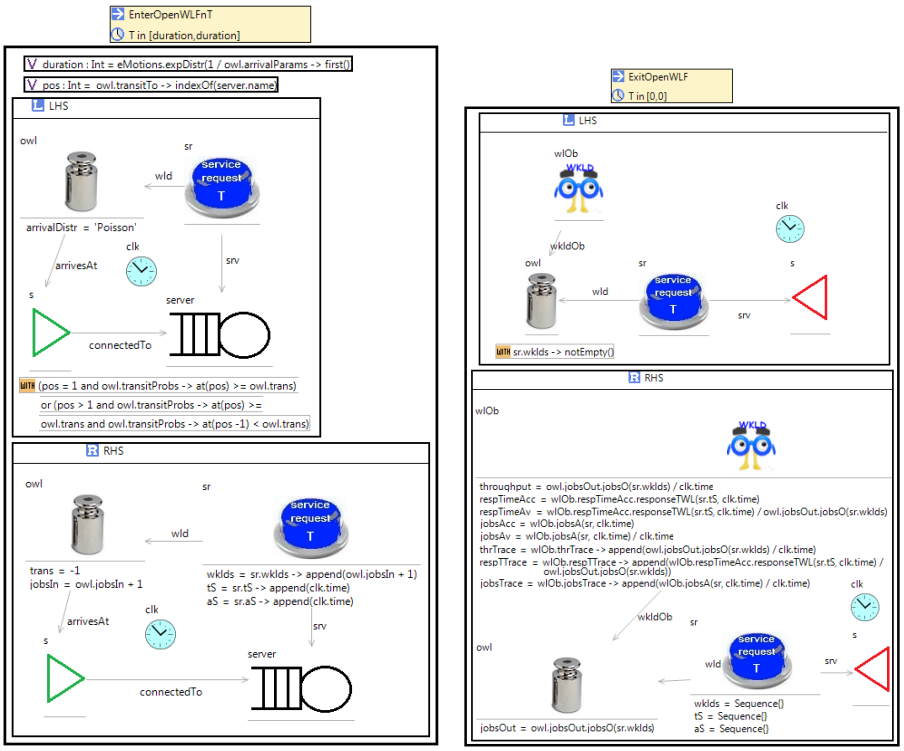
modelos son transformados automáticamente en notación de Maude, tal y como se explicó en la Sección D.2.3. Los resultados devueltos por la herramienta son compatibles con aplicaciones de hojas de cálculo, y tienen la forma que se muestra en la Figura D.18.

En cuanto al criterio de parada en las simulaciones, hemos usado el método de los *long runs* y el método de los *batch means*. En el primero, el usuario ha de especificar el tiempo de simulación, mientras que en el segundo debe especificar el tamaño de las muestras y el índice de varianza.

D.4 Un Enfoque Modular para la Especificación de Observadores

Anteriormente hemos presentado nuestro enfoque para definir observadores que capturen las propiedades no funcionales de los sistemas. Estos observadores se definen a través de un metamodelo y, a continuación, pueden ser incluidos manualmente en las reglas de comportamiento. La adición de observadores en las reglas implica a veces cambiar bastante los modelos de comportamiento, haciendo el modelo del sistema más complejo y difícil de entender, mantener y reutilizar. Un modo de afrontar este problema es el de usar técnicas basadas en aspectos [CB05], donde se proporciona una especificación independiente y genérica de los observadores, que luego se entrelaza con un sistema concreto.

La idea es contar con una librería que contenga la definición de distintos observadores. Definir un observador consiste en crear un DSVL para dicho observador, donde se defina su sintaxis abstracta, su sintaxis concreta y su semántica. Lo ideal si se tienen bibliotecas de este tipo, es que el usuario pueda decidir qué observadores quiere utilizar en su sistema, seleccionarlos, y estos aparecerán automáticamente en las reglas de comportamiento. En este resumen vamos a mostrar un ejemplo en el que definimos un observador individual para monitorizar el concepto genérico de tiempo de respuesta. Después, mostramos cómo el DSVL



(a) Regla EnterOpenWLFnT (b) Regla ExitOpenWLF

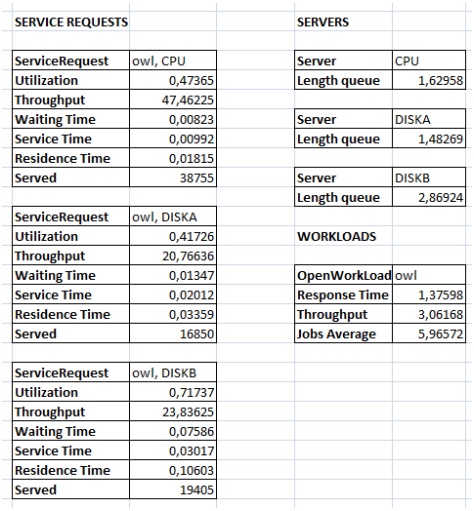
Figure D.17: Reglas para la llegada y salida de paquetes.

definido para este observador se puede entrelazar con nuestro DSVL mostrado en la Sección D.2 para definir cadenas de producción de martillos.

En la Figura D.4 mostramos el metamodelo de nuestro DSVL para definir cadenas de montajes, y la Figura D.6 muestra una regla que modela cómo se ensambla un martillo nuevo. Nuestro objetivo ahora es medir el tiempo de producción de los martillos por parte del ensamblador. Para ello, vamos a definir un observador genérico para medir el concepto genérico de tiempo de respuesta y posteriormente incluiremos el observador en el metamodelo y regla mostrados.

D.4.1 Definición de un Observador Genérico

El primer paso en la definición de un lenguaje para un observador es la definición de su sintaxis abstracta y concreta, que se presentan en la Figura D.19. Podemos observar que en el metamodelo no se define el concepto de tiempo de producción, sino que se define algo llamado tiempo de respuesta (*response time*), que es un concepto más genérico. De hecho, el concepto de tiempo de producción sólo tiene sentido en sistemas de producción. Además, el metamodelo es un modelo paramétrico. Los conceptos de Server, Queue y Request, y las conexiones entre



SERVICE REQUESTS	
ServiceRequest	owl, CPU
Utilization	0,47365
Throughput	47,46225
Waiting Time	0,00823
Service Time	0,00992
Residence Time	0,01815
Served	38755

ServiceRequest	owl, DISKA
Utilization	0,41726
Throughput	20,76636
Waiting Time	0,01347
Service Time	0,02012
Residence Time	0,03359
Served	16850

ServiceRequest	owl, DISKB
Utilization	0,71737
Throughput	23,83625
Waiting Time	0,07586
Service Time	0,03017
Residence Time	0,10603
Served	19405

SERVERS	
Server	CPU
Length queue	1,62958
Server	DISKA
Length queue	1,48269
Server	DISKB
Length queue	2,86924

WORKLOADS	
OpenWorkLoad	owl
Response Time	1,37598
Throughput	3,06168
Jobs Average	5,96572

Figure D.18: Resultados en xQNM

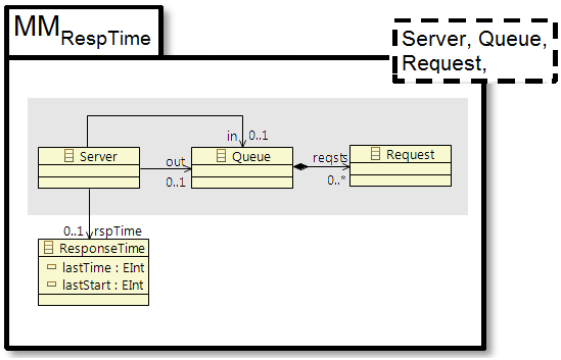
ellos son parámetros del metamodelo, y se han resaltado en gris para diferenciarlos. Estos parámetros se usan para describir situaciones en las que se puede especificar un tiempo de respuesta, pero han de ser mapeados con conceptos concretos de otro DSVL, como vemos en la Sección D.4.2.

La Figura D.20 muestra un ejemplo de una regla de transformación que define una semántica para el observador del tiempo de respuesta. De acuerdo a la regla, si hay un servidor en cuya cola de entrada hay peticiones (al menos una), entonces estas peticiones pasan a la cola de salida en la parte derecha de la regla y, además, se actualiza un atributo del observador para registrar el tiempo actual. La parte de arriba de la regla, resaltada en gris en la figura, es un patrón que describe qué reglas de transformación han de ser extendidas para incluir el observador de tiempo de respuesta. En la figura observamos que esta regla utiliza multiplicidades para expresar que puede haber un número arbitrario de peticiones (al menos una) asociadas a la cola.

D.4.2 Entrelazado de dos DSVL

Una vez contamos con un DSVL de un observador, hay que entralazar el mismo con un sistema concreto. Para ello, hay que definir correspondencias entre ambos DSVL. Estas correspondencias se definen creando un modelo de correspondencias conforme al metamodelo de la Figura D.21. Dicho modelo define correspondencias entre ambos metamodelos y entre las reglas. Respecto a los metamodelos, las correspondecias son:

- Server se corresponde con Assemble, pues queremos medir el tiempo de respuesta de esta máquina.



(a) Abstract syntax (metamodel) (b) Concrete syntax

Figure D.19: Metamodelo y sintaxis concreta para observadores de tiempo de respuesta

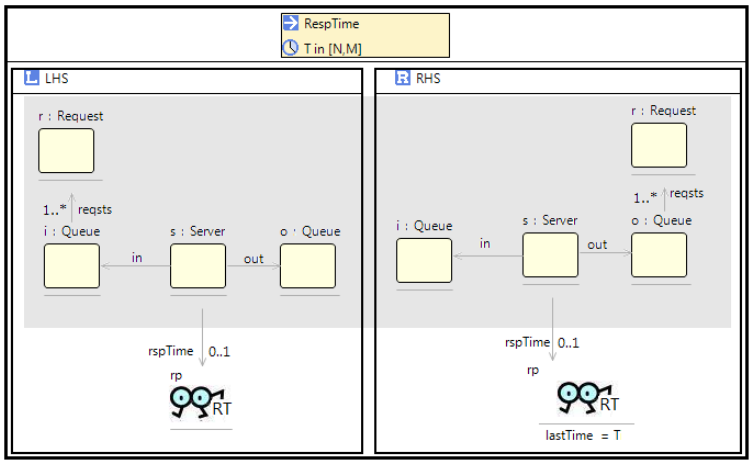


Figure D.20: Regla para RespTime

- Queue con LimitedContainer, ya que los ensambladores tienen objetos de tipo LimitedContained conectados a su entrada y salida.
- Request con Part, pues el ensamblador necesita que haya piezas de martillo para ensamblarlas.
- En cuanto a las asociaciones, las relaciones in y out entre Server y Queue se corresponden con las relaciones in y out entre Machine y Tray, y Machine y Conveyor, respectivamente.

El entrelazado de ambos metamodelos conforme a estas correspondencias produce el metamodelo de la Figura D.22, donde la parte del observador se ha destacado en gris. Se ha añadido el concepto de tiempo de respuesta en el metamodelo, lo que permite usar objetos de tipo ResponseTime en las reglas. Dicho

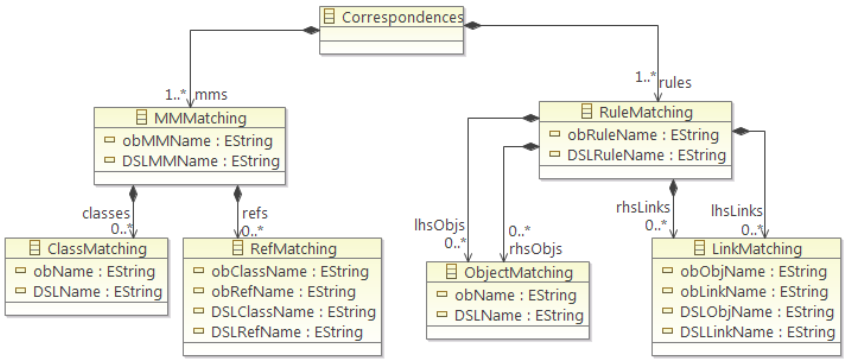


Figure D.21: Metamodelo de correspondencias

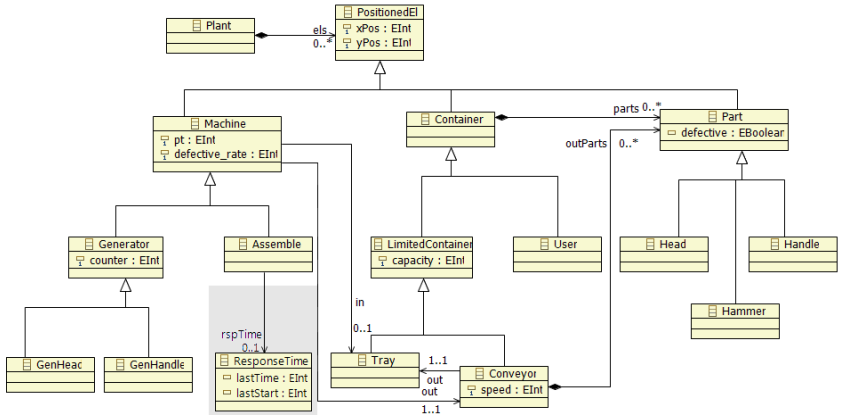


Figure D.22: Metamodelo resultado del entrelazado

entrelazado lo realiza la transformación de modelos escrita en ATL, *WeaveMetaModels.atl*, mostrada en el Apéndice B.

En cuanto a la regla, la vamos a entrelazar con la regla Assemble mostrada en la Figura D.6 con objeto de monitorizar el tiempo de respuesta de los ensambladores. Dicha regla cumple el patrón de la regla RespTime (Figura D.20) del siguiente modo: en la parte izquierda, hay un Server (el ensamblador) con un Queue de entrada (la bandeja) que tiene dos Requests (el mango y cabeza del martillo) y una Queue de salida (la cinta transportadora); en cuanto a la parte derecha, observamos que el patrón se cumple de forma similar. De este modo, los objetos mencionados son los que se relacionan en el modelo de correspondencias. El entrelazado, realizado con la transformación ATL *WeaveBeh.atl* mostrada en el Apéndice B, crea la regla de la Figura D.23.

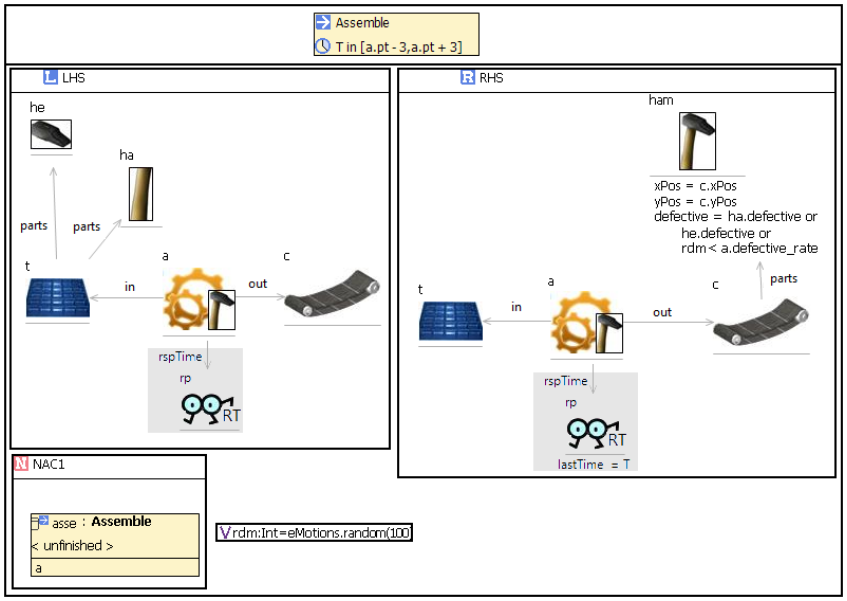


Figure D.23: Resultado de entrelazar las reglas de las Figuras D.6 y D.20

E

Conclusiones y Contribuciones

En este apéndice resuminos las contribuciones aportadas en esta tesis. Estas contribuciones las comenzamos proponiendo un enfoque para evitar un problema que resulta crítico a la hora de construir aplicaciones software o cualquier tipo de sistemas. En el desarrollo de un sistema, cuyo ciclo de vida se presenta en la Figura D.1, los cambios que se requieran realizar en las últimas fases suelen ser muy caros. Además, estos cambios se propagan normalmente hacia atrás (hacia fases más tempranas del ciclo), algo no deseado. Los cambios se han de llevar a cabo porque el sistema no funciona como se esperaba una vez que es desplegado. La razón puede ser que las propiedades no funcionales no se tuvieron en cuenta en las primeras fases del desarrollo. En los Capítulos 3 y 4 presentamos un enfoque para el modelado y comprobación de las propiedades no funcionales de sistemas dinámicos en la fase de diseño. Nuestra propuesta se realiza dentro del campo y las recomendaciones de la Ingería Dirigida por Modelos, donde se usan modelos de dominio específico (en nuestro caso, modelos visuales de dominio específico) como entidades de primera clase.

Hemos propuesto el uso de objetos especiales, llamados *observadores*, que se pueden añadir a la especificación gráfica de los sistemas para describir y monitorizar sus propiedades no funcionales. Los observadores permiten extender el estado global del sistema con las variables que el diseñador quiere analizar al lanzar simulaciones, en la fase de diseño, permitiendo capturar así las propiedades no funcionales de interés. Además, el hecho de que las ejecuciones de las acciones sean ciudadanos de primera clase en las especificaciones en *e-Motions* permite que los observadores monitoricen no solo el estado de los objetos del sistema, sino también sus acciones. En el Capítulo 2 presentamos la herramienta prototipo *e-Motions* para especificar sistemas utilizando lenguajes visuales de dominio específico (DSVL, del inglés, *domain-specific visual languages*). Esta herramienta ha servido como prueba de concepto para nuestra propuesta.

La idea es la de modelar las propiedades no fncionales de los sistemas en la fase de diseño, permitiendo comprobar que se cumplen los requisitos no funcionales en dicha fase. De este modo, el diseño del sistema se puede rehacer hasta que el sistema funciona como se espera en lo que a sus requisitos no funcionales se refiere. En el Capítulo 3 aplicamos nuestro enfoque en dos casos de estudio, un sistema de linea de producción y una red con intercambio de paquetes. Mostramos

cómo se puede monitorizar el comportamiento del sitema a lo largo de toda su vida, definiendo trazas en los observadores, y cómo el cambiar ciertas propiedades del sistema hace que las simulaciones produzcan distintos resultados. La idea es reconfigurar estas propiedades para obtener los resultados no funcionales perseguidos. En el Capítulo 3 nos centramos en propiedades de redimiento, mientras que en el Capítulo 4 nos centramos en la fiabilidad, donde aplicamos nuestro enfoque en el dominio de las redes de sensores inalhámbricas. Modelamos un protocolo para estas redes, llamado DSAP, y explicamos cómo podemos modelar fácilmente variantes del mismo con nuestro enfoque. Además, extendimos el compotamiento del protocolo para modelar situaciones más realísticas. Concretamente, consideramos el consumo de energía en los nodos en *stand-by* y cómo los nodos pueden fallar por circunstancias aleatorias. También realizamos una contribución destacable en el modelado de los sistemas en el Capítulo 4 respecto al modelado en el Capítulo 3, que permite acelerar las simulaciones. Consiste en modelar los objetos que fluyen en el sistema como atributos de clase en vez de como clases en sí mismos. Así por ejemplo, las piezas de los martillos se modelan como elementos en el caso de estudio de la línea de producción, al igual que los paquetes en el ejemplo de la red con intercambio de paquetes. Por el contrario, los paquetes no se modelan como elementos en el caso de la red de sensores, sino que se modelan como atributos de la clase nodo. Esto reduce considerablemente el número de objetos en nuestros modelos, por lo que la identificación de patrones en las reglas de reescritura de Maude se realiza mucho más rápidamente y las simulaciones terminan antes.

Tras presentar nuestra propuesta y aplicarla a varios casos de estudio, la usamos para traducir un formalismo, las redes de colas, a *e-Motions*. Construimos un puente semántico entre el formalismo y un DSVL representado en *e-Motions* y que usa nuestros observadores. El DSVL consiste en un conjunto de reglas genéricas para modelar diferentes tipos de modelos de redes de colas. Dichas reglas se pueden simular para obtener las propiedades de rendimiento de los modelos. En nuestro enfoque, los trabajos, que fluyen entre los servidores, se modelan como atributos, con el objetivo mencionado anteriormente. Además, las reglas se pueden decorar, de modo que se puede extender el comportamiento de las redes. De hecho, mostramos cómo el rendimiento de una red de cola varía si se tienen en cuenta fallos en los servidores. Creamos también una herramienta llamada xQNM, sobre el DSVL, como un plugin de Eclipse. La herramienta permite dibujar modelos conforme a un formato de intercambio de modelos (PMIF), un metamodelo creado como estándar para intercambiar modelos de redes de colas entre herramientas. Concretamente, lo que hicimos fue adaptar PMIF para que fuese conforme a Ecore y poder así integrarlo en Eclipse. A nuestra verisón del metamodelo la hemos llamado ePMIF. Tras dibujar un modelo con nuestra herramienta, éste puede ser simulado para obtener sus propiedades de rendimiento. Se pueden usar muchas distribuciones de probabilidad para los tiempos de servicio y llegada de paquetes. La herramienta también permite importar modelos conforme a PMIF 2,

y exportarlos a notación XML. Esto permite el intercambio de modelos de xQNM con otras herramientas que también soporten PMIF.

En este punto, creemos que merece la pena mencionar una contribución que fue mencionada muy por encima en la Sección 2.1.2, donde presentamos el lenguage de transformación de modelos ATL. Consiste en la dfinición de una semántica formal para ATL basada en lógica de reescritura. Esta semántica se describió en Maude. El uso de Maude como dominio destino aporta beneficios interesantes, ya que permite simular las especificaciones ATL y realizar un análisis formal de los programas ATL. En particular, mostramos cómo nuestras especificaciones pueden hacer uso de Maude para razonar sobre ciertas propiedades de las reglas ATL.

Por último, presentamos una alternativa a nuestro modo de añadir observadores en los DSVL. Pensamos que esta alternativa aumenta la modularidad, flexibilidad y extensibilidad en la definición de observadores. Consiste en definir distintos DSVL para observadores independientemente de ningún sistema concreto. Estos lenguajes se definen de forma paramétrica, donde las clases y referencias paramétricas (en los metamodelos), y los objetos y los enlaces (en las reglas) se instancian luego con conceptos concretos del DSVL de un sistema concreto. Así, la definición independiente de un DSVL para un observador consiste en la definición de un metamodelo paramétrico y un conjunto de reglas de comportamiento paramétricas (puede estar compuesto sólo por una regla). Luego, se realizan dos entrelazados. En el primero, se mapea el metamodelo del observador con el de un sistema concreto. Esto quiere decir que se hace corresponder las clases paramétricas con clases concretas de un sistema, y lo mismo para los enlaces. Tras este entrelazado, las clases que representan observadores aparecen en el metamodelo del sistema concreto. En el segundo entrelazado, las reglas para los observadores se mapean con reglas de un sistema concreto. Así, los objetos paramétricos del primero se mapean con objetos del segundo, y lo mismo para las referecias. Tras el entrelazado, los observadores aparecen en las reglas de comportamiento del sistema.

En el Capítulo 6 presentamos este enfoque, y lo aplicamos para los observadores individuales y generales. Mostramos dos casos de estudio donde se entrelaza la definición paramétrica e independiente de un observador individual con dos sistemas diferentes. De este modo, el mismo observador individual se incluyó en dos sistemas diferentes, tras definir las correspondencias con ambos sistemas. Puesto que los observadores generales normalmente utilizan valores capturados por observadores individuales, mostramos cómo es posible, e incluso recomendable, entrelazar un DSVL de un observador general con el DSVL de uno individual. En este caso, el primero actúa como DSVL paramétrico, mientras que el segundo actúa como DSVL concreto. El resultado de este entrelazado es un DSVL con la definición de dos observadores, donde el general usa valores del individual para realizar cálculos. El DSVL resultante se entrelaza luego con un sistema concreto, de modo que los dos observadores se incluyen a la vez en el sistema.

Bibliography

[AEH06] H.M.F. AboElFotoh, E.S. ElMallah, and H.S. Hassanein. On The Reliability of Wireless Sensor Networks. In *ICC '06*, volume 8, pages 3455 –3460, june 2006.

[AGK+89] A. Aggarwal, K. J. Gordon, J. F. Kurose, R. F. Gordon, and E. A. MacNair. Animating simulations in RESQME. In *Proc. of the 21st conference on Winter simulation (WSC'89)*, pages 612–620. ACM, 1989.

[Alt89] Tayfur Altiok. Queueing Models of a Single Processor with Failures. *Performance Evaluation*, 9:93–102, 1989.

[Alt97] Tayfur Altiok. *Performance Analysis of Manufacturing Systems*. Springer, 1997.

[Ate11] Atenea. e-Motions, 2011. `http://atenea.lcc.uma.es/E-motions`.

[Ate12] Atenea. DSAP and its Reliability in *e-Motions*, 2012. `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/DSAP`.

[BCC+10] Hugo Brunelière, Jordi Cabot, Caue Clasen, Frédéric Jouault, and Jean Bézivin. Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, pages 32–47, Paris, France, June 2010.

[BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[BDIS04] S. Balsamo, A. DiMarco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. on Soft. Engineering*, 30(5):295–310, May 2004.

[BFS03] L. Brenner, P. Fernandes, and A. Sales. MQNA - Markovian Queueing Networks Analyser. In *Proc. of MASCOTS'03*, pages 194–199, Orlando, FL, October 2003.

[BG09] Peter Bazan and Reinhard German. Approximate transient analysis of large stochastic models with WinPEPSY-QNS. *Computer Networks*, 53(8):1289–1301, 2009.

[BGH+05] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba real-time tool suite: model-driven

development of safety-critical, real-time systems. In *Proc. of ICSE'05*, pages 670–671. ACM, 2005.

[BGK+06] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing Applications Using Model-Driven Design Environments. *Computer*, 39(2):33, 2006.

[BGS77] BGS Systems. BEST/1 Product Description, BE77-010-2, January 1977.

[BK94] G. Bolch and M. Kirschnick. The Performance Evaluation and Prediction SYstem for Queueing NetworkS - PEPSY-QNS. Technical Report TR-I4-94-18, University of Erlangen-Nuremberg, Germany, June 1994.

[BMIS04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. on Software Engineering*, 30(5):295–310, 2004.

[BMS06] F. Budinsky, E. Merks, and D. Steinberg. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Longman, Amsterdam, 2006.

[BÖ10] Artur Boronat and Peter Csaba Ölveczky. Formal real-time model transformations in MOMENT2. In *Proc. of FASE 2010*, number 6013 in LNCS, pages 29–43. Springer, 2010.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proc. 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383. ACM Press, 2004.

[BV09] Martin Bravenboer and Eelco Visser. Parse table composition separate compilation and binary extensibility of grammars. In *Proc. 1st Int'l Conf. on Software Language Engineering (SLE 2008), Revised Selected Papers*, volume 5452 of *LNCS*. Springer, 2009.

[CB05] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, Heidelberg, Germany, 2007.

[CDJC08] Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhán Clarke. Model-driven Theme/UML. *Transactions on Aspect-Oriented Software Development*, 2008.

[CDLR12] Eduardo Cañete, Manuel Díaz, Luis Llopis, and Bartolomé Rubio. Hero: A hierarchical, efficient and reliable routing protocol for wireless sensor and actor networks. *Computer Communications*, 35(11):1392 – 1409, 2012.

[CGLM93] Kow C. Chang, Robert F. Gordon, Paul G. Loewner, and Edward A. MacNair. The Research Queuing Package Modeling Environment (RESQME). In *Proc. of the 25th conference on Winter simulation (WSC'93)*, pages 294–302. ACM, 1993.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[CHX+06] O. Chipara, Z. He, Guoling Xing, Qin Chen, Xiaorui Wang, Chenyang Lu, J. Stankovic, and T. Abdelzaher. Real-time Power-Aware Routing in Sensor Networks. In *IWQoS '06*, pages 83 –92, june 2006.

[CMI07] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. Integrating performance and reliability analysis in a non-functional MDA framework. In *Proc. of FASE 2007*, number 4422 in LNCS, pages 57–71. Springer, 2007.

[CW10] Adam Czubak and Jakub Wojtanowski. Lifespan-aware routing for wireless sensor networks. In *Proceedings of the 4th KES international conference on Agent and multi-agent systems: technologies and applications, Part II*, KES-AMSTA'10, pages 72–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[Dan00] Marco Danelutto. Task farm computations in Java. In *Proc. of the 8th International Conference on High-Performance Computing and Networking (HPCN Europe 2000)*, volume 1823 of *LNCS*, pages 385–394, London, UK, 2000. Springer.

[Dat03] Amitava Datta. Fault-Tolerant and Energy-Efficient Permutation Routing Protocol for Wireless Networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[DB78] Peter J. Denning and Jeffrey P. Buzen. The Operational Analysis of Queueing Network Models. *ACM Comput. Surv.*, 10:225–261, September 1978.

[DBL+02] John Drummond, Valdis Berzins, Luqi, William Kemple, Auguston Mikhail, and Nabendu Chaki. Quality of service behavioral model from event trace analysis. In *Proc. of the 7th International Command and Control Research and Technology Symposium*, Quebec, Canada, September 2002.

[DGFD06] Dragan Djuric, Dragan Gasevic, Simon Fraser, and Vladan Devedzic. The tao of modeling spaces. *Journal of Object Technology*, 5:125–147, 2006.

[dLGB+10] Juan de Lara, Esther Guerra, Artur Boronat, Reiko Heckel, and Paolo Torrini. Graph transformation for domain-specific discrete event time simulation. In *Graph Transformations*, number 6372 in LNCS, pages 266–281. Springer, 2010.

[dLV06] Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, 15(3–4):309–330, 2006.

[dLV08] Juan de Lara and Hans Vangheluwe. Translating model simulators to analysis models. In *Proc. of FASE 2008*, number 4961 in LNCS, pages 77–92. Springer, 2008.

[dLV10] Juan de Lara and Hans Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22(3-4):297–326, May 2010.

[DMW98] O. Das and C. Murray Woodside. The fault-tolerant layered queueing network model for performability of distributed systems. In *Computer Performance and Dependability Symposium, 1998. IPDS '98. Proceedings. IEEE International*, pages 132 –141, sep 1998.

[dSeSL00] Edmundo de Souza e Silva and Rosa Leão. The TANGRAM-II Environment. In *Computer Performance Evaluation. Modelling Techniques and Tools*, volume 1786 of *LNCS*, pages 366–369. Springer, 2000.

[Ecl] Eclipse. The Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/`.

[Ecl08] Eclipse. *Graphical Modeling Framework*, 2008. `http://www.eclipse.org/modeling/gmf`.

[EE08] Claudia Ermel and Hartmut Ehrig. Behavior-preserving simulation-to-animation model and rule transformations. *ENTCS*, 213(1):55–74, 2008.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in TCS. Springer, March 2006.

[EHC05] Sol Efroni, David Harel, and Irun R. Cohen. Reactive animation: Realistic modeling of complex dynamic systems. *Computer*, 38(1):38–47, 2005.

[EHKZ05a] Claudia Ermel, Karsten Holscher, Sabine Kuske, and Paul Ziemann. Animated simulation of integrated uml behavioral models based on graph transformation. In *VL/HCC'05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 125–133, Washington, DC, USA, 2005. IEEE Computer Society.

[EHKZ05b] Claudia Ermel, Karsten Holscher, Sabine Kuske, and Paul Ziemann. Animated simulation of integrated UML behavioral models based on graph transformation. In *Proc. of VL/HCC'05*, pages 125–133, Washington, DC, USA, 2005. IEEE Computer Society.

[EPG+07] Pereira P. Ed, Paulo R. Pereira, António Grilo, Francisco Rocha, Mário S. Nunes, Augusto Casaca, Claude Chaudet, Peter Almström, and Mikael Johansson. END-TO-END RELIABILITY IN WIRELESS SENSOR NETWORKS: SURVEY AND RESEARCH CHALLENGES, December 2007. Available from: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.80.2458`.

[FBV+09] Mathias Fritzsche, Hugo Bruneliere, Bert Vanhooff, Yolande Berbers, Frederic Jouault, and Wasif Gilani. Applying megamodelling to model driven performance engineering. In *Proc. of ECBS'09*, pages 244–253. IEEE Computer Society, 2009.

[FG86] A. Federgruen and L. Green. Queueing systems with service interruptions. *Oper. Res.*, 34(5):752–768, October 1986.

[Fie10] Tony Field. *JINQS: An Extensible Library for Simulating Multiclass Queuing Networks V1.0 User Guide*, october 2010. `www.doc.ic.ac.uk/~ajf/Research/manual.pdf`.

[Fis73] G.S. Fishman. Statistical analysis for queuing simulations. *Management Science*, 3(20):363–369, 1973.

[FJA+09] Mathias Fritzsche, Jendrik Johannes, Uwe Aßmann, Simon Mitschke, Wasif Gilani, Ivor Spence, John Brown, and Peter Kilpatrick. Systematic usage of embedded modelling languages in automated model transformation chains. In *Proc. of SLE'09*, number 5452 in LNCS, pages 134–150. Springer, 2009.

[FP98] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[FS05] H. Frey and I. Stojmenović. *Geographic and energy-aware routing in sensor networks*, chapter 12. Wiley, 2005.

[FW97] A. Feldmann and W. Whitt. Fitting mixtures of exponentials to long-tail distributions to analyze network performance models. In *Proc. of INFOCOM'97*, volume 3, pages 1096–1104, April 1997.

[GBC+06] Félix García, Manuel F. Bertoa, Coral Calero, Antonio Vallecillo, Francisco Ruíz, Mario Piattini, and Marcela Genero. Towards a consistent terminology for software measurement. *Information and Software Technology*, 48(8):631–644, August 2006.

[GD07] Daniele Gianni and Andrea D'Ambrogio. A language to enable distributed simulation of extended queueing networks. *Journal of Computers*, 2(4):76–86, June 2007.

[GF99] Manish K. Govil and Michael C. Fu. Queueing theory in manufacturing: A survey. *Journal of Manufacturing Systems*, 18(3):214 – 240, 1999.

[GH09] Holger Giese and Stephan Hildebrandt. Efficient Model Synchronization of Large-Scale Models . Technical Report 28, Hasso Plattner Institute at the University of Potsdam, 2009.

[GHL10] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars. In *Proceedings of the 2010 Workshop on Model-Driven Engineering, Verification, and Validation*, MODEVVA '10, pages 19–24, Washington, DC, USA, 2010. IEEE Computer Society.

[GHV02] Szilvia Gyapay, Reiko Heckel, and Dániel Varró. Graph transformation with time: Causality and logical clocks. In *Proc. of ICGT 2002*, pages 120–134. Springer, 2002.

[GK10] Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[GLSP06] Daniel García, Catalina M. Lladó, Connie U. Smith, and Ramón Puigjaner. Performance Model Interchange Format: Semantic Validation. In *Proc. of ICSEA'06*, pages 47–52, 2006.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 16–27. ACM, 2003.

[GT12] Deepak Goyal and Malay Ranjan Tripathy. Routing Protocols in Wireless Sensor Networks: A Survey. *ACCT12*, 0:474–480, 2012.

[Gun05] Neil J. Gunther. *Analyzing Computer System Performance with Perl::PDQ*. Springer, 2005.

[Gun09] N. J. Gunther. PDQ, 2009. `http://www.perfdynamics.com/Tools/PDQ.html`.

[HAFK11] Tzu-Ching Horng, Nikolas Anastasiou, Tony Field, and William Knottenbelt. LocTrackJINQS: An Extensible Location-aware Simulation Tool for Multiclass Queueing Networks. *Electronic Notes in Theoretical Computer Science*, 275:93–104, 2011.

[HCB00] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8 - Volume 8*, HICSS '00, pages 3005–3014, Washington, DC, USA, 2000. IEEE Computer Society.

[HDSM05] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and M. Mozer. Automating vertical profiling. In *Proc. of OOPSLA'05*, San Diego, California, October 2005.

[HHJZ09] Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE*, 5560:39–82, 2009.

[Hil11] James H. Hill. Data mining execution traces to validate distributed system quality-of-service properties. In *In: Knowledge Discovery Practices and Emerging Applications of Data Mining: Trends and New Domains*, pages 174–197. IGI Global, 2011.

[HLM06] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic graph transformation systems. *Fundam. Inform.*, 74(1):63–84, 2006.

[Hly11] Myron Hlynka. List of Queueing Theory Software. `http://web2.uwindsor.ca/math/hlynka/qsoft.html`, 2011.

[HOT02] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, 2002.

[HSZT00] Christophe Hirel, Robin A. Sahner, Xinyu Zang, and Kishor S. Trivedi. Reliability and Performability Modeling Using SHARPE 2000. In *Proc. of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, TOOLS'00, pages 345–349, London, UK, 2000. Springer.

[HT10] Reiko Heckel and Paolo Torrini. Stochastic Modelling and Simulation of Mobile Systems. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 87–101. Springer, 2010.

[ISO89] ISO. *LOTOS. A formal description technique based on the temporal ordering of observational behaviour*. ISO 8807, 1989.

[ISO99] ISO/IEC FCD 10746-5. *Information Technology – Open Distributed Processing – Quality of Service*, 1999.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality – Part 1: Quality model*. ISO/IEC, 2001.

[JABK08a] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.

[JABK08b] Frederic Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: a model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. Wiley, 1991.

[JB06] Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[JN10] K.A. Jalil and M.H. Nategh. A Composed Energy Aware Metric for WSNs. In *ICCDA*, volume 2, june 2010.

[KAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proc. 8th ACM Int'l Conf. on Aspect-Oriented Software Development (AOSD'09)*, pages 87–98. ACM, 2009.

[Kei62] J. Keilsen. Queues Subject to Service Interruptions. *Annals of Mathematical Statistics*, 33:1314–1322, 1962.

[KH11] Ajab Khan and Reiko Heckel. Model-based stochastic simulation of super peer promotion in P2P VoIP using graph transformation. In *Proc. of DCNET/OPTICS'11*, pages 32–42, 2011.

[KK94] S. Kumar and P.R. Kumar. Performance bounds for queueing networks and scheduling policies. *Automatic Control, IEEE Transactions on*, 39(8):1600 –1611, aug 1994.

[KKW03] Georgia-Ann Klutke, Peter C. Kiessler, and M. A. Wortman. A critical look at the bathtub curve. *IEEE Transactions on Reliability*, 52(1):125–129, 2003.

[Kle06] A. G. Kleppe. 1st European workshop on composition of model transformations (CMT'06). Technical Report TR-CTIT-06-34, Centre for Telematics and Information Technology, University of Twente, June 2006.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a framework for compositional development of domain specific languages. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise*. 2003.

[LCSW07] Yanjun Li, Chung Shue Chen, Ye-Qiong Song, and Zhi Wang. Real-time QoS support in wireless sensor networks: a survey. In *Proc. 7th IFAC Int Conf on Fieldbuses & Networks in Industrial & Embedded Systems (FeT'07)*, 2007. Survey.

[LS 10] LS Computer Technology Inc. SPE·ED, 2010. `http://www.spe-ed.com`.

[LVCA+07] Fernando Losilla, Cristina Vicente-Chicote, Bárbara Álvarez, Andrés Iborra, and Pedro Sánchez. Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach. In Flávio Oquendo, editor, *ECSA*, volume 4758 of *LNCS*, pages 179–194. Springer, 2007.

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[Mar10] Moreno Marzolla. The `qnetworks` Toolbox: A Software Package for Queueing Networks Analysis. In *Proc. of Analytical and Stochastic Modeling Techniques and Applications (ASMTA'10)*, volume 6148 of *LNCS*, pages 102–116. Springer, June14–16 2010.

[Mes92] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MG94] E. A. MacNair and R. F. Gordon. An introduction to the RESearch Queueing Package for modeling contention systems. *SIGSIM Simul. Dig.*, 24:40–70, December 1994.

[MMM02] Daniela Mania, John Murphy, and Jennifer Mcmanis. Developing performance models from nonintrusive monitoring traces. In *Proc. of Proceeding of Information Technology and Telecommunications (IT&T)*. The MIT Press, October 2002.

[MSU+04] S. J. Mellor, K. Scott, A. Uhl, D. Weise, and R. M. Soley. *MDA distilled: principles of model-driven architecture*, volume 88. Addison-Wesley, 2004.

[Mus98] John D. Musa. *Software Reliability Engineering*. McGraw-Hill Inc., New York, NY, USA, 1998.

[ÖM07] Peter Ölveczky and José Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, June 2007.

[OMG01] OMG. Model Driven Architecture - A Technical Perspective, July 2001. Available from: `http://www.omg.org/docs/ormsc/01-07-01.pdf`.

[OMG03] OMG. MDA Guide V.1.0.1, 2003. Available from: `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`.

[OMG04] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. OMG, September 2004. ptc/04-09-01.

[OMG05] OMG. *UML Profile for Schedulability, Performance, and Time Specification*. OMG, January 2005.

[OMG07] OMG. *OMG Systems Modeling Language, v1.0*. OMG, 2007.

[OMG08] OMG. *A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems*. OMG, June 2008.

[OMG10] OMG. UML 2.3.1 Superstructure specification, 2010. Available from: `http://www.omg.org/spec/UML/2.3/Superstructure/PDF`.

[Pag89] Lavon B. Page. *Probability for Engineering With Applications to Reliability*. Computer Science Press, Inc., USA, 1989.

[RD02] P. Rodríguez-Dapena. *Software Safety Verification in Critical Software Intensive Systems*. PhD thesis, Technische Universiteit Eindhoven, 2002.

[RDV09] José E. Rivera, Francisco Durán, and Antonio Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *Proc. of VL/HCC'09*, Corvallis, Oregon (US), September 2009.

[RDV10] José E. Rivera, Francisco Durán, and Antonio Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In *Proc. of WRLA'10*, number 6381 in LNCS. Springer, March 2010.

[RGdLV08] José E. Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *Proc. of SLE'08*, number 5452 in LNCS, pages 54–73, Tolouse, France, 2008. Springer.

[RGF+06] Y. Raghu Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development I*, 3880:75–105, 2006.

[Riv10] Jose E. Rivera. *On the Semantics of Real-Time Domain Specific Modeling Languages*. PhD thesis, Universidad de Málaga, 2010.

[RLPS05] Jerònia Rosselló, Catalina M. Lladó, Ramon Puigjaner, and Connie U. Smith. A web service for solving queuing network models using PMIF. In *Proc. of WOSP'05*, pages 187–192. ACM, July 2005.

[Roc11] Rockwell Automation. Arena Simulation Software, 2011. `http://www.arenasimulation.com/`.

[RRDV07] J. Raúl Romero, José E. Rivera, Francisco Durán, and Antonio Vallecillo. Formal and tool support for model driven engineering with Maude. *Journal of Object Technology*, 6(9):187–207, June 2007.

[RVD09] José E. Rivera, Antonio Vallecillo, and Francisco Durán. Formal specification and analysis of domain specific languages using Maude. *Simulation: Transactions of the Society for Modeling and Simulation International*, 85(11/12):778–792, 2009.

[SBHS06] John M. Slaby, Steve Baker, James Hill, and Douglas C. Schmidt. Applying system execution modeling tools to evaluate enterprise distributed real-time and embedded system QoS. In *Proc. of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 350–362, Sydney, Australia, 2006. IEEE Computer Society.

[Sch86] Herb Schwetman. CSIM: a C-based process-oriented simulation language. In *Proc. of the 18th conference on Winter simulation (WSC'86)*, pages 387–396. ACM, 1986.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '94, pages 151–163, London, UK, 1995. Springer-Verlag.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.

[SEA12] SEALAB Quality Group. WEASEL, 2012. `http://sealabtools.di.univaq.it/toolWeasel.php`.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[SK08] A. Schürr and F. Klar. 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In *4th International Conference on Graph Transformation*, volume 5214 of *Lecture Notes in Computer Science (LNCS)*, pages 411–425, Heidelberg, 2008. Springer Verlag.

[SL04] C. Smith and C.M. Llado. Performance model interchange format (pmif 2.0): XML definition and implementation, April 2004. `www.perfeng.com/paperndx.htm`.

[Smi90] Connie U. Smith. *Performance engineering of software systems*. Addison-Wesley, 1990.

[Smi04] Smith, Connie U. and Lladó, Catalina M. Performance Model Interchange Format (PMIF 2.0): XML Definition and Implementation. In *Proc. of the First International Conference on Quantitative Evaluation of Systems*, pages 38–47, 2004.

[Smi10] Smith, Connie U. and Lladó, Catalina M. and Puigjaner, Ramon. Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability. *Performance Evaluation*, 67(7):548–568, 2010.

[SMM+12] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. *Software and Systems Modeling*, 11(1):111–125, 2012.

[SR02] Rahul C. Shah and Jan M. Rabaey. Energy Aware Routing for Low Energy Ad Hoc Sensor Networks. In *WCNC'12*, March 2002.

[SS04] Ayad Salhieh and Loren Schwiebert. Power-Aware Metrics for Wireless Sensor Networks. *International Journal of Computers and Applications*, 26(4), 2004.

[ST87] Robin A. Sahner and Kishor S. Trivedi. Reliability Modeling Using SHARPE. *IEEE Transactions on Reliability*, R-36(2):186–193, june 1987.

[SW99] Connie U. Smith and Lloyd G. Williams. A performance model interchange format. *Journal of Systems and Software*, 49(1):63–80, 1999.

[SWKS01] A. Salhieh, J. Weinmann, M. Kochhal, and L. Schwiebert. Power Efficient Topologies for Wireless Sensor Networks. In *International Conference on Parallel Processing*, pages 156–163, 2001.

[TBLRV11] Javier Troya, José Bautista, Fernando López-Romero, and Antonio Vallecillo. Lightweight Testing of Communication Networks with *e-Motions*. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs*, volume 6706 of *Lecture Notes in Computer Science*, pages 187–204. Springer Berlin / Heidelberg, 2011.

[TEO05] Yuan Tian, Eylem Ekici, and Füsun Özgüner. Energy-Constrained Task Mapping and Scheduling in Wireless Sensor Networks. In *MASS*. IEEE, 2005.

[Tho06] Kuhne Thomas. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4):369–385, December 2006.

[THR10] Paolo Torrini, Reiko Heckel, and István Ráth. Stochastic Simulation of Graph Transformation Systems. In David Rosenblum and Gabriele Taentzer, editors, *Proc. of FASE'10*, volume 6013 of *LNCS*, pages 154–157. Springer, 2010.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 18–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[Tri02] Kishor S. Trivedi. SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator. In *International Conference on Dependable Systems and Networks (DSN)*, page 544, Bethesda, MD, USA, June 2002.

[TRV09] Javier Troya, José E. Rivera, and Antonio Vallecillo. On the specification of non-functional properties of systems by observation. In *Proc. of NFPinDSML'09*, volume 553, Denver, Colorado, October 2009. CEUR Workshops.

[TRV10] Javier Troya, José E. Rivera, and Antonio Vallecillo. Simulating domain specific visual models by observation. In *Proc. of the Symposium on Theory of Modeling and Simulation (DEVS'10)*, Orlando, FL (US), April 2010.

[TTL09] Long Tran-Thanh and Janos Levendovszky. A novel reliability based routing protocol for power aware communications in wireless sensor networks. In *Proceedings of the 2009 IEEE conference on Wireless Communications & Networking Conference*, WCNC'09, pages 2308–2313, Piscataway, NJ, USA, 2009. IEEE Press.

[TV10] Javier Troya and Antonio Vallecillo. Towards a Rewriting Logic Semantics for ATL. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 230–244. Springer Berlin / Heidelberg, 2010.

[TV11] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10(5):1–29, 2011.

[TV12] Javier Troya and Antonio Vallecillo. xQNM: A domain-specific language to specify and simulate queuing network models, 2012. `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/xQNM`.

[TVDZ13] Javier Troya, Antonio Vallecillo, Francisco Durán, and Steffen Zschaler. Model-driven performance analysis of rule-based domain specific visual models. *Information and Software Technology*, 55(1):88–110, January 2013.

[Val08] Antonio Vallecillo. A Journey through the Secret Life of Models. In *Model Engineering of Complex Systems (MECS)*, number 08331 in Dagstuhl Seminar Proceedings, Germany, 2008.

[VP04] Michel Veran and Dominique Potier. QNAP2: A portable environment for queueing system modelling. In D. Potier, editor, *Proc. of the International Conference on Modelling Techniques and Tools for Performance Analysis*, pages 5–24, 2004.

[WCS00] K. Preston White, Jr., Michael J. Cobb, and Stephen C. Spratt. A comparison of five steady-state truncation heuristics for simulation. In *Proc. of the 32nd conference on Winter simulation (WSC'00)*, pages 755–760, 2000.

[WdMBB06] Patrick Wüchner, Hermann de Meer, Jörg Barner, and Gunter Bolch. A brief introduction to MOSEL-2. In *Proc. of the 13th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems, MMB*, pages 469–472, Nürnberg, Germany, March 2006. VDE Verlag.

[WHRW01] C. Wohlin, P. Höst, P. Runeson, and A. Wesslén. *Software Reliability*, volume 15. Encyclopedia of Physical Sciences and Technology (third edition), 2001.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your models Ready for MDA*. Addison-Wesley, 2nd edition, 2003.

[WTZ10] Christian Wende, Nils Thieme, and Steffen Zschaler. A role-based approach towards modular language engineering. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Proc. 2nd Int'l Conf. on Software Language Engineering (SLE'09)*, volume 5969 of *LNCS*, pages 254–273. Springer, 2010.

[ZJBL08] Tian Zhang, Frédéric Jouault, Jean Bézivin, and Xuandong Li. An MDE-based method for bridging different design notations. *ISSE*, 4(3):203–213, 2008.

[Zsc10] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems. *Software and System Modeling*, 9(2):161–201, 2010.