Master Thesis

In Computer Science, track Software Engineering

a model-based approach for early analysis and prediction of responsiveness

ing. W. Roosenburg

March 14, 2008

Committee	Dr. Ir. B. Tekinerdogan	University of Twente
	H. Sozer MSc.	University of Twente
	Dr. Ir. R. Scholte	Thales Netherlands B.V.
	Ir. M. Glandrup	Thales Netherlands B.V.

TRESE
Twente Research & Education on Software Engineering

THALES

University of Twente
Enschede - The Netherlands

Building a model-based reasoning framework for early analysis and prediction or responsiveness

W. Roosenburg

March 14, 2008

Abstract

A key quality concern in building mission-critical systems is responsiveness; Responsiveness comprises the total end-to-end latency of multiple functional flows that can occur within a system. Thales Netherlands B.V. designs and develops many mission-critical systems, with the "Combat Management System", that provides "command and control" capabilities to naval vessels, as a prime example. This system is being developed at the "System Definition" business unit, that is responsible for defining large, responsive and complex combat software. It is important that system engineers from this business unit can verify the expected performance behavior "on-the-fly" i.e. before the system is actually build. However, within the current system engineering process, performance assessment is performed too late e.g. only after the integration phase. This induces a large financial risk on the developed system, because not meeting performance requirements lead to cost overruns that can result in project failures. Therefore, performance assessment should be considered early during the overall process.

This research thesis presents an integrated performance engineering method that can assess and evaluate responsiveness early; It uses responsiveness-related information that is currently scattered among Thales Netherlands B.V. and their subcontractors. Integration is achieved by defining a model-based process that makes use of current development artefacts (models) and by providing automation to system engineers in the form of modern tools. These tools allow the system engineer to compose systems and to predict the expected performance behavior at the same time.

Structured performance engineering methods requires the existence of metamodels; metamodels describe the structure of systems and can provide a domain-level overview of all important responsiveness concepts that influences the performance behavior of the system. Therefore, this thesis developed two UML metamodels: one for describing the exact structure of systems and one for describing important responsiveness-related quality attributes. By combining both metamodels, a (simple) analytical model can be obtained that can closely estimate the expected responsiveness behavior in practice.
However, analytical models are closed-form solutions: the calculated results represent averages in the form of constants. It is also valuable to verify the certainty of the outcome. Therefore, a simulation-based approach is incorporated within this thesis, as simulations can predict system behavior by using distributions instead of constant values. As a result, the total end-to-end latency of a functional flow cannot be described only by a constant value (e.g. 500ms), but also with a degree of certainty (e.g. a peak latency of 500ms will occur in less than 5% of all cases). This is very important for requirement verification. Model-driven engineering techniques are used in order to "close the gap" between system models and (formal) analysis models with the goal of generating an executable simulation model out of UML system analysis models.

The "proof of concept" tool developed during this thesis uses the IBM Rational Software® product family; It is based on the Eclipse® framework and provides good UML modeling capabilities, an integrated development environment and support for adding and extending existing UML functionalities by means of stereotypes and tagged values. Model-based approaches are supported by the provided JAVA transformation engine. Thales Netherlands B.V. (will) use the UML and the IBM Rational Software® product family for all kinds of system engineering activities.
The tool is available as two IBM Rational Software® plugins: one that provides modeling support to system engineers and one that provides transformations that can calculate and generate a simulation model out of a composed UML system analysis model. This simulation model can be fed into a discrete-event JAVA simulator, that generates histograms with end-to-end latencies that can occur within the composed system. System engineers use this feedback for composing an optimal, responsive system configuration.

Preface

This Master thesis is the written outcome of my final project at the University of Twente, conducted externally at Thales Netherlands B.V. in Hengelo, during the period of May 2007 until March 2008. This is my final work in order to attain my Master's degree.

During my internship, I have been supported by many people. First of all, I would like to thank my supervisors at the University of Twente, Bedir Tekinerdogan and Hasan Sozer, for their constructive criticism. Your help improved the structure and academic maturity of my thesis a lot. Second, I would like to thank Rene Scholte and Maurice Glandrup at Thales. Rene, who was always in for a chat and never unwilling to explain important things twice, because he believed that I would understand it *some* moment in time. Rene, thank you for that! Also a big thanks goes to Maurice, from who I have learned a lot. Your help towards describing the structure of Thales systems and the development of our metamodel is greatly appreciated!

Last but not least, I would like to thank my 'lunch colleagues' at Thales. Boudewijn, Leon, Joris, Gijs, Gregoire, Javier, Sergei and (again) Rene, thanks for all interesting and non-interesting ◡̈ stories and our never-ending discussions about soccer. By the way, the Dutch still rule the French! Hope to see you guys again somewhere in the future.

Wouter Roosenburg
Enschede - March 14, 2008

.

'The limits of my language mean the limits of my world'
Ludwig Wittgenstein

Contents

List of Figures

List of Tables

List of Abbrevations

AWS	Above Water Systems
CMS	Combat Management System
COTS	Commercial Off The Shelf
CS	Combat System
CSCI	Computer Software Configuration Item
CSDN	Combat System Data Network
DDS	Data Distribution Service
DFD	Data Flow Diagram
MDA	Model Driven Architecture
MOF	Meta Object Facility
MS&SE	Modeling, Simulation and Synthetic Environment
NRT	Non Real-Time
OC	Operator Console
OMG	Object Management Group
OPENSPLICE	Open Implementation for SPLICE
RSA	Rational Software Architect
RT	Real-Time
S14S2	SPLICE1 For SPLICE 2
SigMa	Signaal Modular Architecture
SPE	Software Performance Engineering
TACTICOS	TACTical Information and Command System
TTNL	The Thales Naval Netherlands
UML	Unified Modeling Language
UWS	Under Water Systems

Part I

Setting the context

Chapter 1

Introduction

Nowadays, developing and maintaining highly responsive systems is a complex task. This is due to the fact that the total responsiveness of a system is determined by many factors that could all possibly influence each other. Add up the still growing size of an average software system and its increasing complexity and it should be clear that analyzing and predicting the responsiveness of a system is not an easy task.
However, current research in the field of "performance analysis" is improving and shows that analyzing and predicting system behavior within a specific domain can be achieved. A current trend is to analyze and predict system behavior during the early development phase, that is, before the system is actually built and integrated, because fixing performance issues earlier on is cheaper than if the system was already built. Although current literature does not show yet a common ontology [Cor05], applying domain specific knowledge to performance models can result in a realistic understanding of the final system behavior.
This thesis will make a contribution into the field of "performance analysis". It captures domain specific knowledge (models) from a real industry case (developed within Thales Netherlands B.V.) and applies performance analysis techniques in order to come up with an early performance prediction judgment. Software specification and automation are key issues here, as they largely determine the successful appliance in industry.

1.1 Context

The Thales Netherlands B.V. Above Water Systems (AWS) department develops and integrates complex combat system software for naval vessels. An example of such a complex combat system is the "Combat Management System" (CMS). The CMS is a "mission-critical" system: it has to meet a number of performance-related quality requirements such as reliability, robustness and real-time responsiveness, in order to perform well. Not meeting these requirements can result in disastrous consequences, such as not being able to respond to an incoming threat on-time, which is totally unacceptable in a combat situation.
Within AWS, a specific discipline is "Lead System Integration" (LSI). This field is focused on managing the complexity of the CMS. It encompasses the areas of "operational performance analysis", "system performance analysis", and "proof of concept demonstrators". A key concern within LSI is performance prediction during the early development phase. However, practice shows that during the early development phase it is difficult to predict the exact system behavior of a combat system, because there is a lack of good tooling support that aids system engineers with performance modeling and performance analysis models. As a consequence, predicting or simulating the system behavior in a structured manner is difficult.

This master thesis is not the first research assignment carried out on this subject within Thales Netherlands B.V. Previous research, performed by [Hoo06] [Hor06] and [Hor07] has led to initial understanding in how to reason about performance and how to model it. Both research studies were aimed at improving the system performance prediction of the CMS used within Thales Netherlands B.V.

The first research study that was conducted on this subject, was performed by Niek Hoogma in 2006 [Hoo06] and was mainly concerned with building a simulator that could simulate system behavior at functional flow level. Its contribution was that activity diagrams could be annotated using the OMG UML SPT Profile [Gro05]. These annotations resides at flow level and contains information about e.g. latency and can be fed into a discrete-event simulator. This simulator was built in JAVA based on the freely available Desmo-J [Des07] simulation library. The output of the simulator was a report that estimated the average latency of a system. Ivo Ter Horst [Hor06] improved the work of Hoogma by conforming more to the SPT profile, because Hoogma introduced new concepts that made his method incompatible with the original SPT profile.
Ter Horst continued the performance research with investigating the possibility of adding "budget information" to resources. The notion of a "budget" got introduced, which describes the resource demand of a hardware or software entity. This was expressed using parameters, such as the number of CPU MIPS available and amount of RAM available. Its final contribution was the development of a layered system model and the possibility to add budget information at several levels: lower-layered budgets could be dependent of higher-layered budgets. By using an UML deployment diagram, the total resource utilization could be calculated.

1.2 Problem statement

Thales Netherlands B.V. is interested in early performance analysis and performance prediction, that is, predicting system behavior without actually possessing a running system. This in order to prevent expensive changes necessary later in the process. An important quality factor that largely determines the final system behavior of a combat system is responsiveness: the "end-to-end latency".
Using a systematic method that supports early performance analysis and prediction is important for Thales Netherlands B.V. for a number of reasons:

1. *Changes made to the system due to a lack of responsiveness are expensive* - the earlier (performance) bottlenecks are detected, the better. Changes made later on the process are much more expensive than if the change was made earlier on, especially if this leads to architectural or requirements changes.

2. *Using a systematic performance engineering method prevents scattering* - in some cases, performance engineering information is scattered among subcontractors. This information, which contains complex formulas, is stored in Microsoft Excel spreadsheets and are exchanged between Thales Netherlands B.V. and its subcontractors in order to limit the resource usage. Generally, a subcontractor is hired for implementing a piece of the system, but the available resources available for the implementation-piece that a subcontractor has to develop is restrained, because a naval vessel does not have unlimited space available for i.e. placing hardware cabinets (that can provide additional resources). Space on a naval vessel is also costly.
   The usage of Microsoft Excel spreadsheets leads to scattering of performance information (different versions, different metrics used) and managing this information centrally is hard. Integrating performance engineering methods within the regular development cycle of the system leads to consistency among performance metrics used, and prevents the ad-hoc incorporation.

3. *Preventing uncertainty during the design phase* - a well known question among system engineers that has been asked many times during the design phase of the CMS is: 'does this application or extension still fits on my system?'. Answering this question becomes hard when the development of the system is split up into different departments and subcontractors, and nobody has an overview of the (emergent) performance behavior of the system.

4. *Assessing the extensibility and flexibility of the system* - important quality factors within system engineering are extensibility and flexibility. Extensibility means how well the system is prepared for possible changes. Changing the radar type for instance can lead to an increase in resource usage, if the radar works at a higher revolution speed. Using a strucured performance engineering method can assess the extensibility of the system. Flexibility means how well the

system performs under different environments. Patrolling on the open sea generally means less possible targets that have to be tracked than if the naval vessel was patrolling in a harbor. Using a structured performance engineering method can also provide an estimation of the flexibility of the system.

This thesis continues the research already initiated by Thales Netherlands B.V. that led to first understandings and prototypes that could analyse and verify performance at different design abstraction levels. The main problem of the previous, proposed solutions were that they are standalone tools, and thus not well integrated within the system development process used within Thales Netherlands B.V. Integration into the development process and the availability of automated tools are key success factors in order to successfully apply performance engineering within industry.
Current system engineering methods use UML based methods in order to develop and design systems. UML models are supported by modern tools applied within industry and can be easily extended by meta-models and profiles. This provides adaptability within models and compatibility among tool vendors. Furthermore, Model Driven Engineering (MDE) will be used in the future within Thales software and system engineering and therefore, a model-driven or model-based approach is preferable.

The above information leads to the initial *problem statement*, which is defined as

What ingredients are necessary to be able to develop a model-based performance prediction framework that addresses real-time responsiveness and integrates within the current system engineering process used within Thales Netherlands B.V. ?

1.3 Contribution

This contribution of this thesis encompasses *methods*, *tools* and *simulation*. The method that this thesis proposes is presented in chapter 4, were changes to the system engineering process that are necessary to incorporate performance engineering, are described. In order to describe the structure of systems and the performance model, an UML meta-model is developed. This is presented in section 5.
The tools that are developed are described in chapter 7. Here, an overview is given of the structure of the proof-of-concept tool. The tool also supports the evaluation of models by means of simulation. Model building and evaluation is described in chapter IV.
However, the solution to the problem that this thesis tries to solve also touches on a number of general problems currently found within literature. They are the following:

1. *This thesis closes the gap between system models and analysis models* - many performance analysis and evaluation techniques use formal models, such as Queuing Networks and Markov models, as their underlying analysis method. However, integrating performance modeling activities within regular system development does not imply that system engineers have to model their architecture as a queuing network (this is called the "gap" between system models and analysis models). One contribution of this thesis is the development of an UML meta-model that explicitly defines what system components can be modeled by system engineers and what kind of performance information can be attached to them. Furthermore, modeltransformations are used in order to transform annotated system models into simulation analysis models. As a result, the "gap" between the two is closed. This is a concept that is acknowledged in current literature [Gro07] [PS02] [Sch06] but how to exactly do it remains unclear.

2. *This thesis uses domain specific information in order to evaluate system models* - when composing system models and attaching performance information, the models needs to be evaluated. Thales Netherlands B.V. already possesses many measurements taken from real running systems that can help with evaluating the system. The developed prototype tool is able to specify and change the calculations / metrics that the tool performs on the model in a flexible way. This stimulates *separation of concerns*: the system engineer composes the system and the performance engineer provides the necessary performance metrics.

3. *This thesis uses simulation to verify the results* - using domain specific (analytical) information for evaluating system models is not enough in order to obtain a realistic understanding of the final behavior of the system. Analytical approached can give an estimation of how the system performs in practice, but it are effectively closed form solutions, which means that the results are constant numbers. In order to obtain a better understanding of the dynamic behavior of the system, simulation is incorporated. Distributions are used that describe the value and probabilities of events that can occur within the CMS, and therefore the outcome is also a distribution. Simulation increases the performance analysis quality: it can verify the outcome of the analytical solution and it provides extra information, such as the probability that a certain latency can occur within the system. This is also important for verifying non-functional requirements of the system.

1.4 Approach

This thesis investigates the development of an integrated performance engineering method that analyses and predict the responsiveness of the Combat Management System, without having the running system available. As a proof of concept, a tool will be developed that supports performance analysis and performance prediction during the early development phase. To verify the usability of the tool, a case study will be developed that uses real software artefacts produced during the system development life-cycle in order to estimate the final system behavior.
In order to successfully perform the above activities, research questions have been defined that have to be answered in order to find a solution to the problem. The main research question is defined as follows:

What process- and product changes are necessary in order to be able to perform early analysis and prediction of responsiveness for the Combat Management System (CMS) used within Thales Netherlands B.V. ?

Subquestions to the main research questions are:

- What is responsiveness?
- Are there general methods and frameworks available within current literature that address performance and responsiveness?
- What is the impact of responsiveness on system design?
- How can consistency be achieved between system design and performance models?
- How to built a consistent set of tools that supports the models used within a typical system engineering process?
- How can obtained performance results be verified?

This thesis starts with explaining the research context and investigating the current system development process within Thales Netherlands B.V. It examines all the produced artefacts during this process. It also describes a number of problems that are present in the current process because of the lack of early performance analysis and prediction.
After that, a number of key concepts that are necessary to understand performance engineering, responsiveness and performance modeling are investigated. Software Performance Engineering [Smi02] is a structured method that shifts performance assessment towards an earlier development phase. A reasoning framework [BIKM05] provides a concrete “body of knowledge” that can be used by non-experts in order to understand how to reason about performance. Responsiveness is the main performance factor considered within this research thesis, because investigating more quality attributes is not feasible within the planned timeframe, and outside the scope of this thesis.

After the required background information is gathered, a model-based reasoning framework is developed. This framework seperates process from product: it provides an overview of the individual steps that the system- and performance engineer take in order to compose the desired system

and evaluate the responsiveness. It also describes the general architecture of the solution and the models needed for using the solution.

The final solution comprises a "two phase approach":

1. The system engineer composes a system, using components based on a meta-model developed within this thesis. Furthermore, he/she adds budget information to system components, using a meta-model that describes the possible budget information that the system engineer can add. As a result, the feasibility of the composed system can be calculated, which basically results in a yes/no answer.

2. When the outcome of the feasibility calculations are within expectations (the system "fits" on available hardware), the system engineer develops behavioral models(UML activity diagrams) of key system components, or components that most likely contain performance bottlenecks. Those models are annotated with latency information and middleware related performance information. Using middleware related formulas (obtained by Thales Netherlands B.V. when measuring a running system) leads to an initial estimation of end-to-end latencies. As a verification step, simulation is used to verify if the obtained results are within the expected range of the chosen distributions. The output is a histogram which shows the tabulated frequencies of the overall latency.

Model transformations are used within the developed tool (together with a dedicated parser) in order to calculate responsiveness and to generate a simulator performance model out of the (annotated) system model. Meta-models are developed because the solution should not only work for one situation specifically, but for every system that is an instance of the system meta-model, annotated with information that is an instance of the quality attribute meta-model. A case study is developed to verify the solution and the usability of the tool.

Finally, the solution is evaluated; Conclusions are drawn and recommendations for the future are given.

1.5 Outline

This thesis is split up into four parts, where each part contains one or more chapters. The first part, 'Part I - Setting the Context', introduces the problem statement and research questions. This is described in the current chapter, **chapter 1**. **Chapter 2** investigates the research context and the development process as used within Thales Netherlands B.V. **Chapter 3** contains background concepts, such as software performance engineering, reasoning frameworks, UML and profiles and responsiveness.
The second part, 'Part II - Concept development and Solution design', sketches the outline of the model-based solution that is developed using the knowledge and theory obtained in Part I. **Chapter 4** describes the contents of the model-based reasoning framework that is developed within this thesis. It describes the process as well as the product. **Chapter 5** contains the developed metamodels: the system metamodel and the quality attribute metamodel. **Chapter 6** contains an exact description of the evaluation procedure: how can responsiveness be calculated and how is it measured?
The third part, 'Part III - Implementation', describes the actual implementation of the solution. In **chapter 7** a proof of concept, DESIDE (DEcision Support In Design Environments), is developed which contains an implementation of the metamodels and the evaluation procedures. It can evaluate the feasibility of a system in a modern UML design environment: IBM Rational Software Architect. Furthermore, it can generate a simulation model and produces outputs in the form of a histogram.
Finally, 'Part IV - Evaluation' reflects back to all the work that is being done during this research thesis and gives recommendations for the future. **Chapter 8** contains the conclusion and provides an overview of work that still has to be done in the future within the "performance analysis and performance modeling" field.

Chapter 2

Thales Netherlands B.V. and the research context

Thales Netherlands B.V. uses System Engineering for the development of their CMS. Generally, System Engineering comprises a set of tasks and activities that have to be executed successfully in order to design and develop complex systems. These activities are referenced to as SIMILAR. This chapter provides an overview of the SIMILAR activities, relates them to Thales Netherlands B.V. and observes possible problems and bottlenecks that can occur when executing these activities during the system engineering process.

2.1 The Combat Management System

As stated in chapter 1, the Above Water System (AWS) department develops and integrates the Combat Management System (CMS), which is specifically targeted for use on naval vessels. The CMS keeps track of objects that possibly could be a threat, but also friendly objects are tracked. The possible range and number of objects that could be tracked depends mainly on the radar type that a customer wants to use.
In order to obtain a better overview of the performance aspects of the CMS, consider figure 2.1 below.

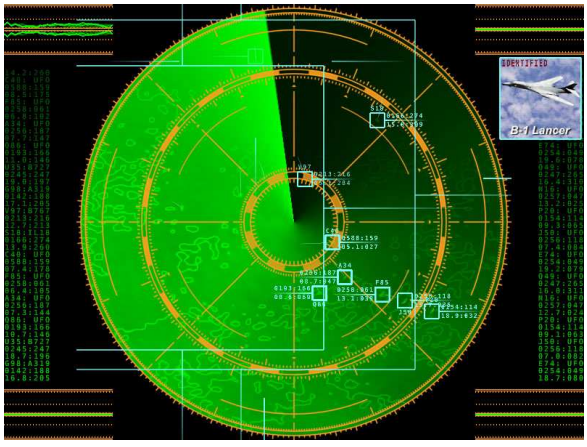

Figure 2.1: Example radar console

The above figure is an example of a (surveillance)radar console. An important measure of a radar its *revolution rate*, which is expressed in *rpm*, and means the number of full rotations completed in one minute around a fixed axis. Also, an often used measure is the *hertz rate*, which is expressed in *Hz*, and means the number of revolutions per second.
Assume that the radar console depicted in figure 2.1 is hooked to a radar with a rpm of 12.

This means that a full rotation will occur every 5 seconds. If there are 300 objects within range, effectively $\frac{300}{5} = 60$ *plots* a second are produced. A plot is the most atomic data item possible: it is one measurement in a polar coordinate system, which means that it comprises only a radius and an angle. After the plots are gathered, they are *correlated*, which basically means that the system decides whether a plot belongs to a certain object (e.g. another vessel) after performing numerous processing steps. If it does (it does not have to, a radar also picks up garbage) a *track* is produced. A track belongs to a moving object, and is basically nothing more than a vector with additional information. Whereas a plot was only one measurement, a track encompasses more information such as moving speed and identification information (friend or foe).
The most important aspect that determines the overall performance of the system is *the rate at which system tracks are produced*. This rate is dependent of the number of incoming objects, because they determine the plot- and track rate and impose a significant load on the system: the amount of resources necessary (e.g. CPU and RAM) can increase dramatically when the number of incoming targets grows.
High-load scenarios can also occur. When incoming objects are identified as foes, they get a higher update ratio which increases the system load. They are "tracked" by specialized trackers, which are sensors of other radars. These trackers also increases the load on a system, because much more data due to a higher update ratio, has to processed.

2.2 The development process

System engineering is an interdisciplinary field of engineering, because it is applied to the development of complex systems (such as combat systems), where many (specialized) disciplines deliver a part of the system. The final system is thus the sum of its parts.
According to [Inc07], system engineering is defined as: *an engineering discipline whose responsibility is creating and executing an interdisciplinary process to ensure that the customer and stakeholder's needs are satisfied in a high quality, trustworthy, cost efficient and schedule compliant manner throughout a system's entire life cycle.* This process comprises a set of tasks [Inc07], which is depicted in figure 2.2.

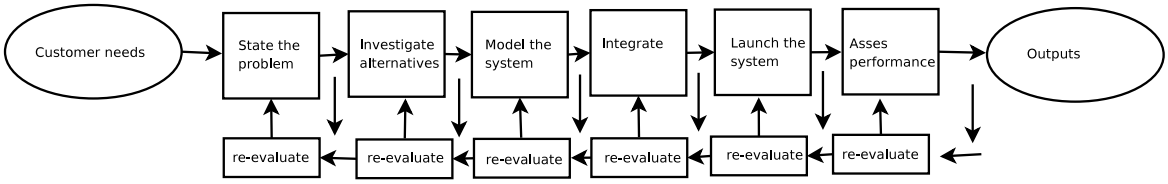

Figure 2.2: The SIMILAR tasks

The tasks depicted in 2.2 are referenced to as SIMILAR. The main point of the SIMILAR tasks, and the engineering discipline that uses it, is that the system should be treated as a whole. This means that phases of the system development life-cycle are not only executed in a sequential manner, but also in parallel. As a result, after every phase there has to be the possibility of evaluating and to go one step back in the process, if needed. Figure 2.2 shows this. Table 2.1 provides a more detailed description of the SIMILAR tasks, as depicted in figure 2.2.

Task name	*Description*
State the problem	Description of the top-level functions that the system must perform
Investigate alternatives	Creation of alternative designs based on cost, performance, schedule and risks
Model the system	Modeling of the system and possible alternatives
Integrate	Define interfaces between subsystems
Launch the system	Run the system and produce outputs
Assess performance	Measure system performance and perform tradeoff analysis
Re-evaluate	Continuous feedback-loop in order to modify the system if necessary

Table 2.1: Description of the SIMILAR tasks

The development process of the CMS, as used within Thales Netherlands B.V., has many similarities with the SIMILAR tasks. In order to categorize SIMILAR, there are a number of so called "key area's" defined that are important to system engineering in general. They are described in table 2.2.

Key area name	*Description*
Operations	How the system operates in practice
Performance	How the system performs in the real world
Test	How to test the system
Manufacturing	How to physically build the system
Cost	How to stay within budget when developing and how to determine the total development cost of the system
Training and support	How to train people to use the system and giving accurate support
Disposal	How to determine the lifecycle of the system

Table 2.2: Description of key areas used within SIMILAR

It is clear that the SIMILAR tasks almost touches on every aspect of system engineering. In this thesis, the performance aspect is only "key area" that is being considered. The next subsequent sections describes how SIMILAR is implemented within Thales Netherlands B.V. and what (performance related) problems arises from using it.

2.2.1 State the problem

The first step in the system engineering process is to get a thorough understanding of the customers' wishes. After customer needs are investigated, which basically means that the customer specifies which sensors and actuators he/she wants, the CMS options are determined. This includes whether or not to include a training component, or external communication services. When this is finished, response times are defined for various combat scenario's.
These response times comprise the end-to-end latency of a system, e.g. the time that passes between object detection and presenting the track to the operator on the screen. If the response times are determined, corresponding budget information is established. This budget information comprises maximum CPU resource usage, maximum RAM resource usage and the maximum latency usage. Sometimes the customer imposes extra constraints on the system, if high-load scenarios are a key issue. This budget information is also used to restrain the resource usage among subcontractors and to get a better overview of the feasibility of the system.

2.2.2 Investigate alternatives

Investigating alternatives means creating alternate designs for comparison. Alternate radars, sensors and actuators can be choosed and an initial performance analysis can be performed. Multiple performance scenarios are calculated using scenarios. At the end, the customer chooses the scenario that fits the requirements best.

2.2.3 Model the system

Modeling the system and possible alternatives is a giant step in the system engineering process. Usually, when a new component or module of the CMS is developed, a developer starts with *domain level* modeling in the form of a conceptual model, which is a *high level overview* of the system to develop. As an addition to this conceptual model, a functional flow diagram is constructed, which is basically an UML activity diagram (the object oriented equivalent of a Data Flow Diagram) where the most important *high-level* functional flows are defined, as well as performance requirements(defined in the first step of the system engineering process). This basically boils down to defining a maximum latency between processes, as described in section 2.2.1.
The second abstraction-level that is being distinguished during the early development process is the *architectural* level. At this stage, the architectural model and the relational model are the

most important artefacts. The architectural model is an UML-like diagram where architectural concepts are shown. These concepts are derived from the relational model, which is a more fine-grained model of the conceptual model.
The third abstraction level is the *design* level. At this level, the detailed design is constructed, as well as the 'topic model'. The topic model is the lowest information-model design artefact, and it describes the publishers and subscribers related to a certain topic as well as the topic itself and its relations to other topics. An overview of all models and their relations is depicted in figure 2.3.

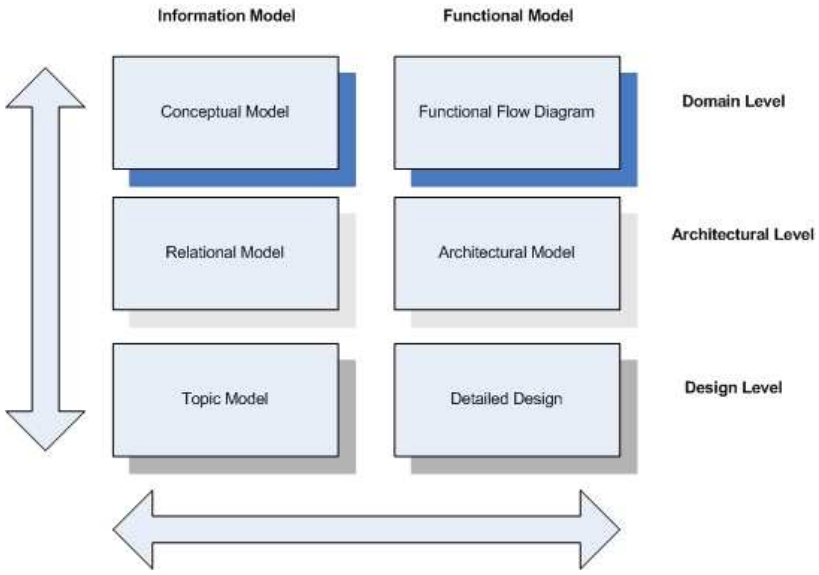

Figure 2.3: Model artefacts produced during the (early) design phase

Modeling the system and its behavior

The produced models shown in figure 2.3 are related to each other with regards to the typical development process used within Thales Netherlands B.V, which is mainly data-oriented. There is a *horizontal relationship* between the produced models in the sense that they are complementary: one cannot be produced without the other and vice versa. There is a *vertical relationship* between the models, because a model that is positioned on a lower level is a refinement of its parent.
Furthermore, it is important to realize that the different models produced during the three phases describes the *system behavior* in a static and dynamic way: the architectural (class) models describes the architectural entities of the system, which is mainly a static diagram, whereas the functional flow diagrams show the dynamic behavior of the system at flow level.

Modeling performance of the system

Ideally, assessing system performance should start at the earliest SIMILAR task (stating the problem): when requirements are gathered alongside with stakeholders. A direct result of assessing system performance at the end of the development process, is that the cost to repair an requirement error during the integration phase is many times higher than during the requirement / development phase. A report from the Standish Group even shows that incomplete or unclear requirements are on top of the list when it comes to project failures and cost overruns [Gro95].

Traditionally, there exist two types of requirements:

1. Functional requirements - they determine *what* the system should perform
2. Non-functional requirements - they determine *how* the system should perform

Performance is strongly related to non-functional requirements, and determines the overall quality of a system. However, quality is a "broad" definition and there are several ways to interpret and

measure the quality of a system. The ISO standardization group [Iso07] has defined a framework for evaluating the software-quality of a system, which is also known as the ISO-9126 standard. It comprises the areas [ESS07] described in table 2.3.

Quality category	*Description*
Functionality	Are the required functions available in the software?
Portability	How easy is to transfer the software to another environment?
Maintainability	How easy is to modify the software?
Efficiency	How efficient is the software?
Usability	Is the software easy to use?
Reliability	How reliable is the software?

Table 2.3: Description of the ISO-9126 quality categories

When investigating the SIMILAR tasks and the ISO definition, there is a clear similarity between the two: performance impacts various phases of the system engineering process and it should be taken into account from the earliest phase, when requirements are gathered.
However, traditional performance assessment techniques assume that there is a running system available. Shifting performance assessment to the early development phase automatically means that other methods need to be used (such as analytical and simulation-based techniques) in order to predict the performance of a system, instead of measuring.
This need for early performance prediction also raises the issue of verification and validation: How do we know that obtained results are correct, even when there is not a running system available? According to the IEEE Standard Glossary of Software Engineering Terminology [IEE07], verification is defined as 'the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase', whereas validation is defined as 'the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements' [IEE07]. Verification focuses on showing that the output of a certain phase conforms to the expected input and validation focuses on showing and proving that the results that are obtained are actually correct.

2.2.4 Integrate

The integration of the individual components is performed by system integrators. All components and modules from subcontractors and internal departments are put together in order to compose the final system. Key issues concern "the emergent properties of the system": how does the system respond and perform when all the components have to work together? As a result of this, the system is throughly tested (using simulated scenarios or spreadsheets) and if it does not perform within expectation, the system engineers need to go back to the drawing board.

2.2.5 Launch the system

After the system is integrated, it is launched: it is actually deployed on a naval vessel and operational testing is performed. Real running measures are obtained and functional requirements are assessed.

2.2.6 Assess performance

After the system is launched and operational testing has been performed the performance of the system is assessed. If a non-functional required is not met, system engineers have to fix at the running system itself, if possible. A drawback of this manual process is that every version of the CMS deployed at naval vessels is different, maintenance is hard and the impact is massive when performance issues are not fixable when the system is deployed.

2.3 The development product

The Thales Naval Netherlands (TTNL) Systems Definition department is responsible for the system definition of the CMS. The primary goal of the CMS is managing the combat system. A Combat System can be defined as a set of resources consisting of machinery and human operators. Examples are sensors, weapons and information systems. These resources determine the total fighting capabilities of a naval vessel.
The CMS controls the CS by means of providing Operator Consoles (OC) to operators. An operator console provides detailed on screen information and helps the operator with the decision-making process. A typical CMS scenario would be displaying an unidentified object on the OC. As a consequence, the operator is given the choice of tracking the object or identify it as a friend or neutral object. The (abstract) overview of the CS and its relation to the CMS is given in figure 2.4.

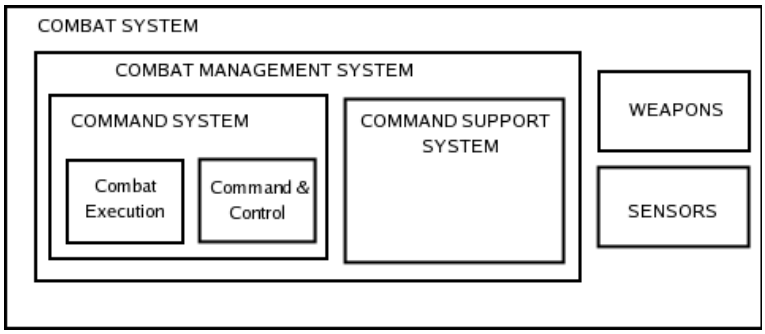

Figure 2.4: Overview of the Combat System and the Combat Management System

Figure 2.4 shows that the CS encompasses the CMS. The CS encompasses the physical sensors and effectors, whereas the CMS only contains items such as middleware, operator consoles and interfaces to sensors and effectors. This clearly draws the system border between the CS and the CMS.
The CMS is further subdivided in two parts: the Command System and the Command Support System. The command system is decomposed into two subsystems: Combat Execution and Command & Control (C&C). Combat execution is responsible for war-fighting: a typical scenario in this context could be firing a 30mm gun. Command and Control is responsible for the situational awareness component of a naval-vessel: sensor data is gathered in order to evaluate possible threats and their corresponding actions. Real-time responsiveness is a key issue concerning both subsystems.
Finally, The Command Support System is responsible for mission planning or looking into historical data. This subsystem is not under stringent performance requirements.

2.3.1 TACTICOS

An instance of the CMS is TACTICOS (TACTical Information and Command System) [Gee07]. TACTICOS consists of various subsystems, such as sensor-services, situational awareness and Combat Warfare. TACTICOS is a modular, highly distributed CMS that is aimed at real-time performance. Different versions (and capabilities) of TACTICOS are spread-out on different type of ships. In order to get a better overview of TACTICOS, TACTICOS components and their relations between them, the architecture is described according to two different views(architectures) of the 4+1 view model according to Kruchten [Kru95]. This comprises the logical architecture and the development architecture.

2.3.2 The logical architecture

According to [Kru95], the logical view is defined as *supporting the functional requirements - what the system should provide in terms of services to its users.* An overview of TACTICOS and its most important modules(as depicted as *components*) is given in figure 2.5, as taken from [Gee07]. A description of the TACTICOS modules is given in table 2.4.

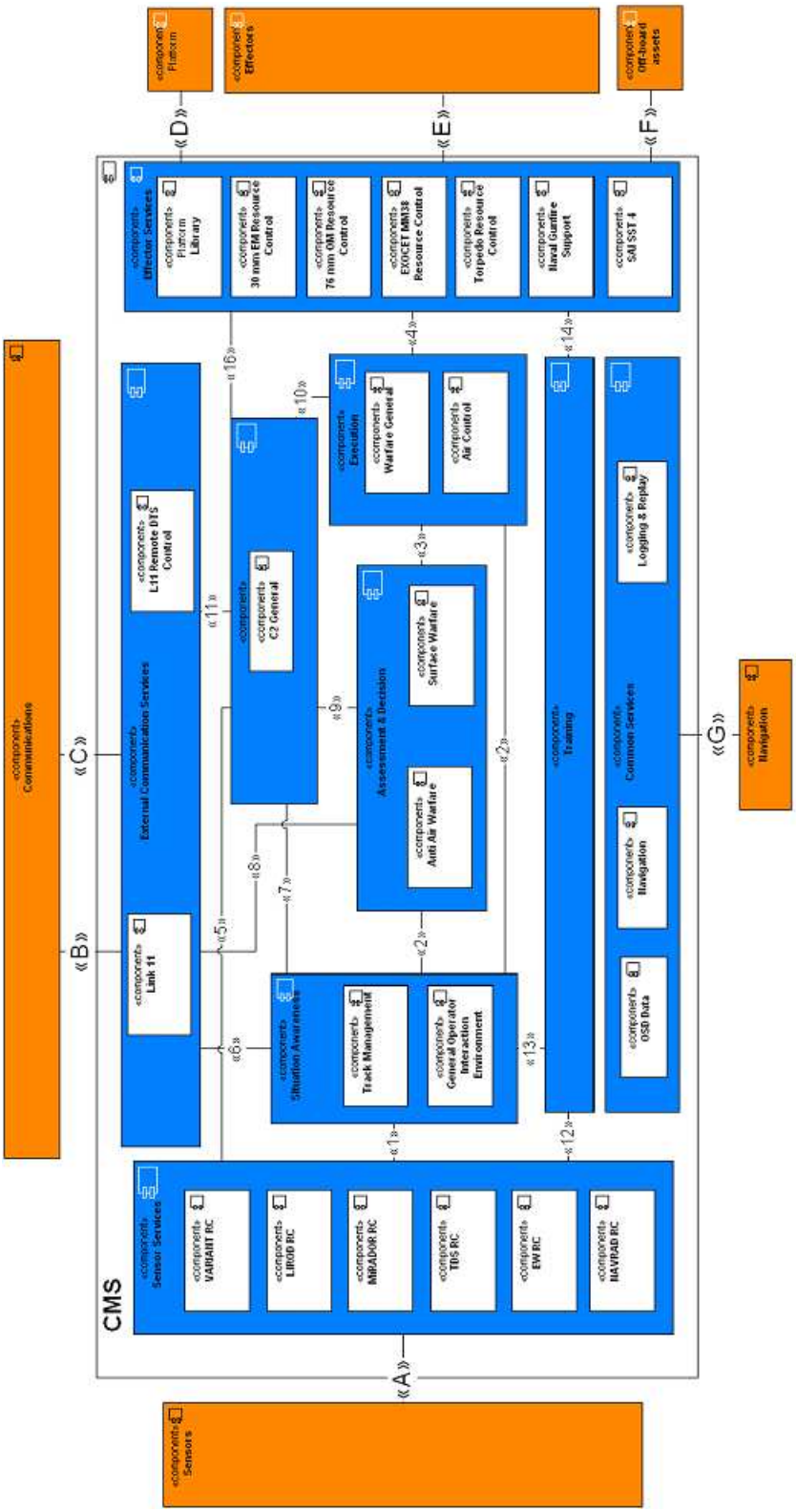

Figure 2.5: TACTICOS logical architecture [Gee07]

Module	*Description*
Sensor services	Responsible for generating sensor track data and sensor measurement data
Situation Awareness	Responsible for interpreting and relating track data
External Communication Services	Provides communication to the outside world
Mission Planning and Control	Responsible for mission planning
Assessment & Decision	Provides warfare capabilities
Execution	Responsible for executing engagement orders. Also provides kill assessment
Effector / Asset Services	Responsible for engagement plans. Monitors the effectors
Training & Simulation	Provides simulations for training purposes
Common Services	Provides various data and functions, e.g. it keeps track of ship movements and gives insight into historical data.

Table 2.4: Description of TACTICOS modules

Table 2.4 (and figure 2.5, show that the CMS is a complex system. It consists many subsystems, each with their own performance requirements. This thesis will focus only on the 'Situation Awareness' part.

2.3.3 The development architecture

The development architecture focuses on, according to [Kru95] *the actual software module organization on the software development environment.* The (basic) development architecture is shown in figure 2.6.

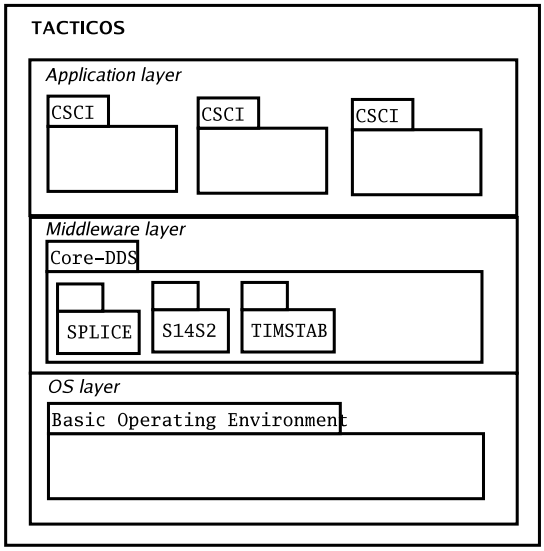

Figure 2.6: TACTICOS development architecture

The application layer

The applications reside at the top level, layer 3. Applications are mainly represented by Computer Software Configuration Items (CSCI). A CSCI has one specific task, which could be gathering sensor data or refreshing operator data. The typical CSCI behavior is collecting information from a source entity(such as a radar), do some processing and write the processed data back into the system using the middleware database. If needed, CSCI's can exchange information between each other using its SPLICE/DDS middleware via a publish-subscribe mechanism.

The middleware layer

The middleware layer contains Core-DDS. Core-DDS (Core-Data distribution Service) is the general architectural view on middleware. SPLICE, as depicted on figure 2.6, is for instance a component of Core-DDS, because the usage of other types of middleware is also possible, such as CORBA. Core-DDS can be seen as a number of connected SigMA(Signaal Modular Architecture) nodes interconnected via CSDN (Combat System Data Network). Multiple SigMA nodes, as well as standard COTS (commercial-off-the-shelf) nodes connected via CSDN form 'the intended system', which in this thesis is represented as the CMS. COTS nodes are nodes with a non real-time purpose (NRT), whereas SigMA nodes covers the need for real-time purposes (RT).
One of the most important components of Core-DDS is SPLICE (also known as SPLICE/DDS), because of its real-time distributed character. SPLICE/DDS is a middleware layer that provides location transparency and real-time responsiveness. It also contains an 'in memory' database, which can be used to store and retrieve high-volume data. The lowest data-entity in this database is called a 'topic'.
Nowadays, SPLICE/DDS is a combination of the OMG standard for 'Data Distribution Service' [Gro05] and its real-time publish-subscribe implementation (SPLICE). Thales Netherlands B.V. originally developed SPLICE and proposed it to the OMG as an official DDS standard. When it finally was accepted, a number of changes were made by the OMG that made the Thales SPLICE version incompatible with the OMG DDS standard. As a result, OpenSplice was developed to provide backwards compatibility via the S14S2 components(Splice 1 for Splice 2). Along with that, Thales Netherlands added TIMSTAB in its middleware layer. TIMSTAB provides time synchronization along a distributed network.
At some point, Thales Netherlands B.V. decided to outsource OpenSplice to PrismTech[Pri07]. The combination of OpenSplice and DDS is now referenced to as SPLICE/DDS. SPLICE/DDS is under stringent performance requirements because all inter-CSCI communication depends on this component. The typical SPLICE/DDS usage is depicted in figure 2.7.

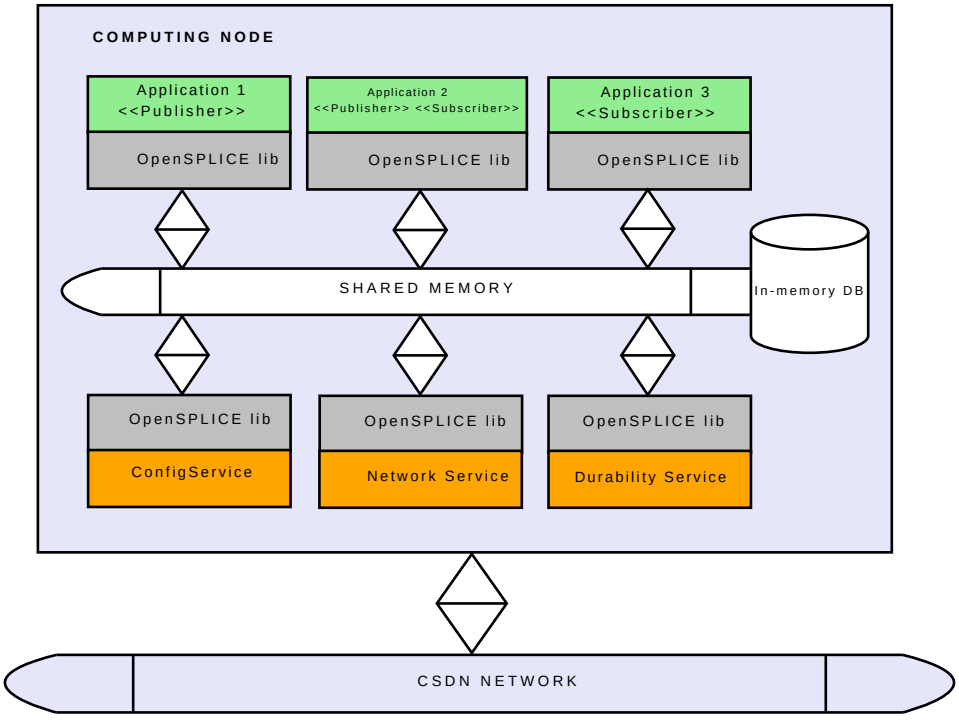

Figure 2.7: Splice infrastructure overview [Pri07]

The Operating System layer

The operating environment resides at the bottom layer of figure 2.6. This layer contains the 'basic operating environment', which provides basic access to the hardware of a SigMa node. A complete overview of a SigMa node is given in figure 2.8.

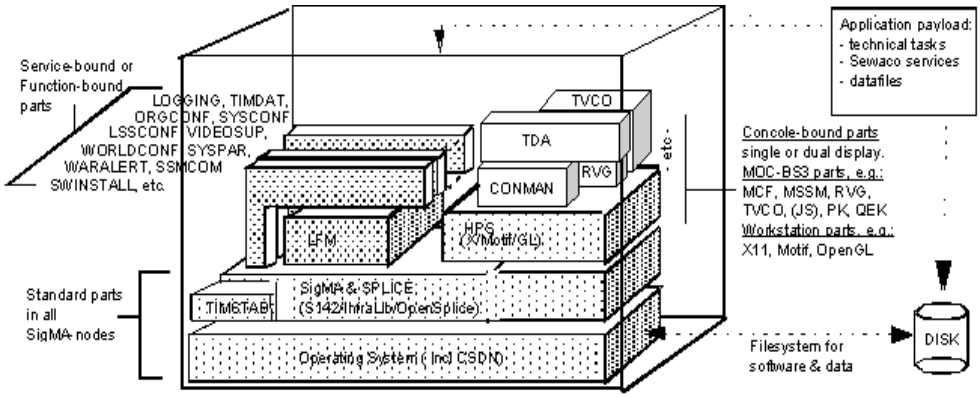

Figure 2.8: Architectural view of a Core-DDS SigMA node [Tha07]

2.4 Problems and observations within the development process and product

There are a number of problems that come forward after investigating the development process of the CMS and studying the produced development artefacts:

1. *Performance assessment is performed late in the development process* - which leads to high costs when repairing performance related issues.
2. *Resource (budget) information is scattered along the organisation* - because it is performed using Excel spreadsheets. Version management of those spreadsheets is hard.
3. *There does not exists a (meta) model of the system* - despite of all the models that are produced during the design phase. In order to develop a structured approach for early performance prediction, a model of the system is a first prerequisite. Only if such a model is available, responsiveness parameters can be attached in order to evaluate the performance of the system.

Also a number of important observations were gathered:

1. *Responsiveness of the system is mainly determined by middleware overhead* - because TACTICOS leans heavily on its SPLICE middleware infrastructure, the total responsiveness of the system is mainly determined by the time that data arrives at producer databases and the time that data is read by consumers. This is an important observation when evaluation procedures has to be built.
2. *There is a lack of automated tools for performance evaluation* - although tools do exists, they do not meet the specific needs of Thales Netherlands B.V. Also these tools are not able to interpret models that are produced during the design phase of the CMS.

2.5 Conclusion

The early identification of performance bottlenecks is a key issue during system definition: if it turns out that the operational performance of a running system is not satisfactory, expensive changes are necessary in order to fix issues due to non-functional requirements. Therefore, the goal of early performance analysis and prediction is to identify those bottlenecks early during the overall process: before the system is actually built.
Due to the incorporation of subcontractors and stringent performance requirements, a structured method needs to be developed that integrates system modeling and performance assessment. This approach also "forces" system engineers to explicitly assess the (early) performance of the system, and it prevents the ad-hoc approach. A model-based UML approach is preferable, but unfortunately, a good (meta)model of the system does not exist (yet).
Performance assessment within this thesis is restricted to analyzing the responsiveness of the CMS targeted at the SPLICE middleware infrastructure, because the end-to-end latency of the CMS is mainly determined by its SPLICE usage. Therefore, methods aimed at evaluating responsiveness of the CMS should largely be focused towards analyzing the SPLICE middleware overhead.

Chapter 3

Early model-based performance prediction

Responsiveness is the key quality concern considered within this research thesis. For Thales Netherlands B.V. responsiveness comprises the "end-to-end" latency of functional flows that can occur within a system. However, current literature in the field of "operational performance analysis" contains other definitions of responsiveness (and its related quality factor, scalability). This chapter investigates both approaches, describes the commonalities and variability, and points out which definition is the most usable for early, model based performance prediction of responsiveness.
The performance analysis and prediction process contains two important activities: "system modeling" and "performance modeling". System modeling means composing the intended system, whereas performance modeling means evaluating the performance of the intended system, by means of adding performance related information onto system models. System modeling leads to system models whereas performance modeling leads to system analysis models. This chapter describes the methods and tools necessary for integrating both activities within a model-based framework, with responsiveness as the key quality factor.

3.1 Designing responsive and scalable systems

Generally, there are two important dimensions to software performance: *responsiveness* and *scalability*. Responsiveness is 'the ability of a system to meets its objectives for response time or throughput' [Smi02] whereas scalability is 'the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases' [Smi02]. Both definitions are closely related, and in both definitions is "response time" the commonality.
Thales Netherlands B.V. is very interested in responsiveness as a software performance quality factor, because (as described in the previous chapter) responsiveness mainly determines the overall performance of the CMS. A non-responsive system will lead to system delays, which are unacceptable in combat situations. Furthermore, scalability determines how well a system "scales" towards a more, demanding environment (e.g. more targets to track). Because the CMS is a "middleware intensive" system, *this thesis will focus only the responsiveness of its middleware component.*

Thales Netherlands B.V. defines responsiveness as the *end-to-end* latency of functional flows that can occur within a system. Basically, this is a time-based definition that can be expressed and measured in seconds(s) or milliseconds (ms). A clear, non-functional requirement using this definition could be that the time between determining a target and the generation of the corresponding track may not exceed 50ms. This is an example of one functional flow of the system. However, this functional flow can be decomposed if necessary and subrequirements may be defined, that (obviously) has to satisfy the general 50 ms requirement mentioned above.

Designing responsive and scalable systems that have to satisfy those functional-flow requirements is not an easy task. Two key design issues that affect responsiveness and scalability are, according to [RVH95]:

1. *Composition* - the policy for allocating software components to "Operating System" processes
2. *Distribution* - the policy for distributing "Operating System" processes across nodes in a network

Composition has a major impact on design and thus has to be considered early. Moreover, a good composition enables flexible distribution. Distribution mainly deals with configuration for specific target environments. Both composition and distribution have a major impact on responsiveness and scalability. Composition can restrict shared data (impacts responsiveness) and distribution can reduce the number of processes that runs on a node (impacts scalability). Whenever designing responsive and scalable system, both key design issues have to be considered carefully. Currently, composition and distribution policies are both implemented in the Sigma / Splice middleware component, as described in section 2.3.3.

3.1.1 Model building

To design responsive and scalable systems, models are needed in order to compare alternative composition and distribution policies. According to [RVH95], software performance models always have two kinds of parameters:

1. *Structure parameters* - Describes the blocking relationships between software components, using parameters such as the number of visits or service time requirements.
2. *Resource demand parameters* - Describes the used resource usage during the execution of software components, using parameters such as the consumed CPU MIPS and RAM memory.

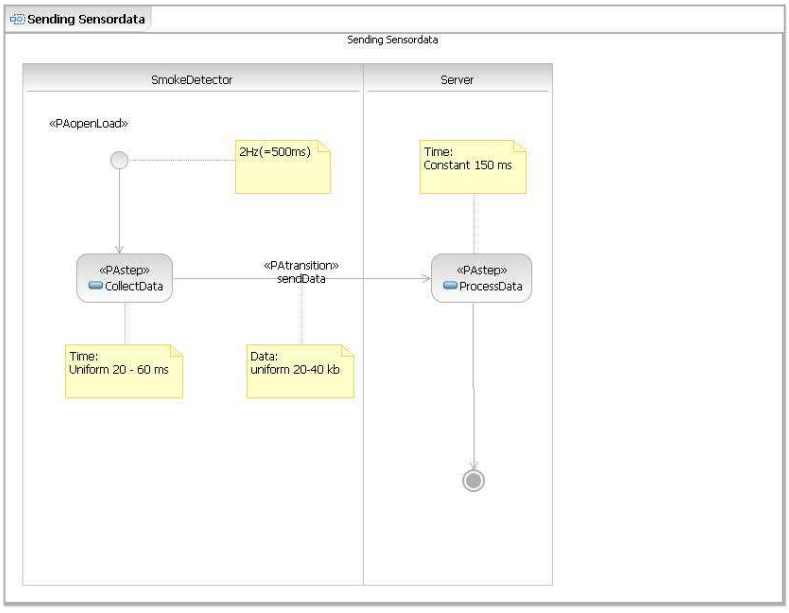

Figure 3.1: Example software performance model

Figure 3.1 shows an example of a software performance model that makes use of an annotated UML activity diagram. The swimlanes (server, smokedetector) represent two resources. On each resource, a process is executed which takes a number of milliseconds (ms) to execute (the service time requirement). This can (possibly) be characterized using a distribution, because the service time requirement does not have to be the same due to runtime characteristics of a system (e.g.

queuing delays, interrupts). A process can also exchange data with another process, but has to take the throughput requirements into account. The throughput requirements (e.g. the maximum throughput capacity) can be drawn on the same diagram, or on another diagram (the UML deployment diagram).

3.1.2 Model Evaluation

There are a number of techniques available that can help to evaluate the built performance model in order to obtain the desired performance characteristics. This is useful, because it gains understanding of the behavior of the system and it is necessary to compare alternate designs.
Generally, there are two approaches for evaluating performance models:

1. **Analytical analysis** this closed-form, mathematical approach "solves" an analytical model in a deterministic way. Obtained performance characteristics are usually "real" or "average" values. Examples of this approach are Stochastic Process Algebras [BBS02] or Layered Queuing network models [PS02].
2. **Simulation-based analysis** this approach "simulates" a performance model in order to build a stochastic, probabilistic model. It relies on probabilities and distributions, instead of constant values. Therefore, it can describe the occurrence of events with a certain probability.

Analytical approaches have the advantage that they calculate an "exact" result and that the algorithms are usually fast, but this approach also has a drawback: it uses constant numbers as input and outputs are real numbers, or averages. When building a responsiveness model, an initial intuition in the form of "the responsiveness of process x is y ms" is not enough for validating requirements, also because all assumptions, such as the number of incoming targets and tracks, are based on constant numbers. A better (realistic) approach would be that some information can be obtained such as "the responsiveness of process x is in 95% of all the cases under y ms". This is more valuable for system engineers, because requirements are also stated that way and it eliminates the constant numbers, because functions with a probability have to be used now. A simulation approach, which uses distributions instead of constant numbers seems a good candidate. However, a drawback of using a simulation-based approach is that the quality of the obtained results are as good as the simulation model itself.
However, the two approaches can also be used in conjunction with each other. When a system engineer expects that there may exists certain bottlenecks within a composition, he may use an analytical approach for obtaining a first intuition of the expected system performance. More detailed information can be added to specific system components that can be expected troublesome, and a simulation analysis can be executed. If the result of this simulation does not conform to the initial intuition, the composition needs to ga back to the drawing board.

Analytical approach

This section will elaborate on a number of topics found within "operational performance analysis", found within [Buz76], in order to provide an overview of an abstract approach of modeling systems and responsiveness. It describes *Little's law* and the Utilization law that can be used when determining the responsiveness of an abstract system.

The most important (and fundamental) *law* in determining responsiveness is *Little's law* [Lit61]. This law states that the average number of requests in a system must be equal to the product of the throughput of the system and the average time spent in the system by a request (the end-to-end latency of a request). Little's law needs three quantities, so if two are known the third one can be obtained as well. Little's law can be derived from the *utilization law*, that can be used in order to determine the utilization of a resource. Combing the two rules together results in the average (average)queuing delay of a request at a certain time. [Buz76].

Consider the abstract system depicted in figure 3.2 below.

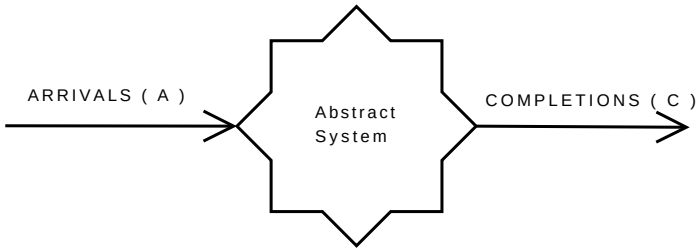

Figure 3.2: Abstract model of a system
[Buz76]

The following quantities could be observed from figure 3.2, during observation time T (the total length):

- A, the number of arrived requests
- C, the number of completed requests

The above observations leads to the following quantities:

Definition 1 *λ, the arrival rate:* $\equiv \frac{A}{T}$

Definition 2 *X, the throughput:* $\equiv \frac{C}{T}$

If the system consists of a single resource, the *busy-time* can also be measured:

Definition 3 *B, the length of time that the resource was observed to be busy*

The previous quantities leads to the following definitions:

Definition 4 *U, the utilization:* $\equiv \frac{B}{T}$

Definition 5 *S, the average service requirement per request:* $\equiv \frac{B}{C}$

With the above definitions, the *utilization law* can be derived. Utilization is defined as $\frac{B}{T}$, which algebraically equals $\frac{C}{T}\ \frac{B}{C}$. Because of $\frac{B}{T} \equiv U$, $\frac{C}{T} \equiv X$ *and* $\frac{B}{C} \equiv S$, the utilization law gives:

$$The\ utilization\ law : U = XS \tag{3.1}$$

The utilization (of a resource) is the product of the throughput (of a resource) and the average service requirement (of a resource). Suppose that a process writes data onto a disk with 75Hz, 75 times a second, with each write requiring 0.0125 seconds of disk. As a result, the utilization law calculates that the utilization of the disk must be 93,75%.

The utilization law is a useful analysis technique, because it is fast and does not make many assumptions. In fact, the utilization law is a special case of a more general law: Little's law [Lit61]. Little's law uses some additional definitions.

Definition 6 *N, the average number of requests in the system:* $\equiv \frac{W}{T}$

Definition 7 *R, the average system residence time per request:* $\equiv \frac{W}{C}$

In the above definitions, W is defined as the accumulated time between the arrival and the completion of a request, usually measure as #requests per minutes or #requests per second. If two requests in ten seconds were observed, W would 20 (which should be read as 20 accumulated request-seconds).
With the above definitions, *Little's law* can be derived. Because algebraically $\frac{W}{T} = \frac{C}{T}\ \frac{W}{C}$ and $\frac{W}{T}$ $\equiv$ N, $\frac{C}{T} \equiv$ X and $\frac{W}{C} \equiv$ R, Little's Law gives:

$$Little's\ Law : N\ = XR \tag{3.2}$$

This means that the average number of requests in a system is equal to the product of the throughput of a system and the average time spent by a request.

Now, suppose that the *average* number of requests of the whole system is 30. The disk is serving 75 requests a second. Then Little's Law calculates that the average time spent by a request must be $\frac{30}{75} = 0.4$ seconds. From the Utilization law, the service time requirement S was .0125, so the average queuing time of a request must be .03875 seconds. (0.4 seconds was devoted to time spent at both queuing and receiving service, of which .0125 was devoted to receiving service, so the difference must be the queuing time).

Simulation-based approach

A (computer) simulation of a system is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied how it works. By changing variables, prediction may be made about the behavior of the system. The difference between an analytical (mathematical) solution and a simulation is that the analytical approach tries to find a solution that enables the prediction of the behavior of the system by a set of parameters and initial conditions. However, when the system has a high degree of complexity, finding an (solvable) analytical closed form solution is most of the time not possible. Instead, simulations are used in order to find a reasonable approximation of the solution.
Simulations can be divided in three categories, according to [Pag84]:

1. **Monte Carlo simulation** - Monte Carlo simulation is a method by which an inherently non-probabilistic problem is solved by a stochastic process; the explicit representation of time is not required.

2. **Continuous simulation** - Within continuous simulation, the variables are continuous functions, e.g. a system of differential equations.

3. **Discrete event simulation** - If value changes to program variables occur at precise points in simulation time (i.e. the variables are "piecewise linear"), the simulation is discrete event.

Also variations of the mentioned above can occur. A **hybrid** simulation can use an analytical submodel within discrete event simulation, whereas **gaming** simulation contains components of all the three simulation categories above.

Previous research, performed by [Hoo06] and citehor06 led to the development of an *discrete event* simulator, based on the freely available DesmoJ [Des07] simulation library. The simulator takes as input an (annotated) UML activity diagram, see figure 3.1, and produces outputs in the form of a graph that shows end-to-end latencies and throughputs of scheduled activities and consumed resources. The simulation differs from analytical models in the sense that input parameters are not constant values, but *distributions*. Examples are "uniform" distributions or "poisson" distributions. Furthermore, the simulation runs for instance 1000 times, with continuously generated random numbers (within the bounds of the chosen distributions). In the end, this results in a histogram of end-to-end latencies and throughputs.

3.1.3 Formal models or a model-based approach?

During this literature study, the question raised if a formal-model approach was preferable or that a pragmatic, model-based approach would be the way to go. The model-based approach was chosen over the formal-model approach, because of the following three reasons:

1. *Not all responsiveness information is available during the modeling phase* - When system engineers develop compositions of the system, not all responsiveness-related information is available during the modeling phase. Usually, a system engineers knows from previous experience how well certain components of the system will respond. Only those components that likely will be the bottleneck are modeled in great detail. And it is exactly those components that will be simulated. Formal models have the disadvantage that in order to be able to generate a detailed performance verdict, all complex factors have to be known in advance. Within Thales Netherlands B.V. this is usually not the case.
2. *Formal models are too complex to use for system engineers* - System engineers only think about system concepts at system level. Formal models are therefore not suitable to be directly used by system engineers, because they focus too much in depth on low-level concepts.
3. *Formal models do not integrate well within current design artefacts* - Currently, a lot of responsiveness information is stored within Excel spreadsheets and a lot of system compositions are already developed. The proposed method has to integrate within current development artefacts and this rules out the usage of new, formal models.

However, not using formal models directly does not imply that they will be left out. Simulation-approaches (and the previously developed simulator) uses "queuing networks" as the underlying formal methodology. However, the advantage of using model-based techniques is that other models can be generated. Simulation will be incorporated in this thesis, and simulation models will be generated out of system models.

3.2 The model-based approach

Using clean, UML models when describing systems, structure and performance have a number of important advantages over "formal" approaches, as good models possess all of the following characteristics [Sel04]:

- *abstract* - they emphasize important aspects
- *understandable* - expressed in a form that is readily understood by observers
- *accurate* - faithfully represents the modeled system
- *predictive* - can be used to answer questions of the system
- *inexpensive* - much cheaper to construct and study than the modeled system

Figure 3.3 illustrates this by using a simple example: it shows that models can visualize architecture much better than program-code. Box1 only contains program code and extracting the architecture from code is much harder than if a model was available. Box2 shows the actual architecture in the form of a model, which is immediately clear after a first glance.

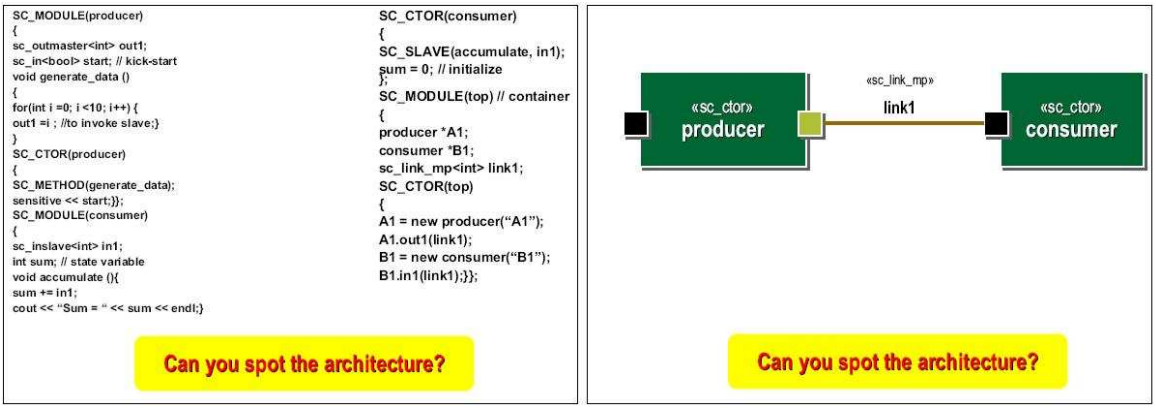

Figure 3.3: The need for working model-based [Sel04]

Using well defined, clean and structured models is the "heart" of *model-driven engineering*, that address issues such as platform complexity and domain specific concepts [Sch06], by means of abstraction and automation.

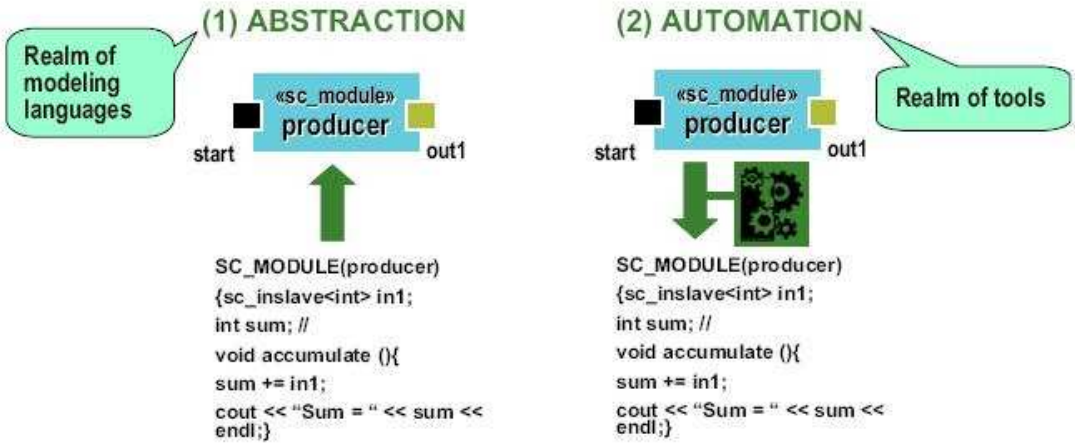

Figure 3.4: Abstraction and automation in Model Driven Engineering [Sel04]

Model driven engineering (MDE) also combines the following [Sch06]:

1. *Domain specific modeling languages* - that formalize the structure, behavior and requirements within particular domains. They are described using metamodels which define concepts and the relationships between them.

2. *Transformation engine and generators* - that analyse certain aspects of models and produce various artefacts such as program code or other models. This helps to ensure consistency among requirements, models and implementation.

The approach presented in this thesis is not a strict model-driven engineering approach. Usually, within MDE, model transformations are executed in order to refine the models further, according to the CIM, PIM, PSM analogy. The ultimate result is to generate working code with a certain degree of complexity, only using clean (intermediate) models. This approach uses MDE in order to develop an integrated performance engineering method. Models are used in order to provide abstraction and automation to performance engineering. The goal of using MDE within this approach is to make the system model *executable*, that is, being able to simulate the composed system.
However, there exists a gap between system models and formal system analysis models(such as queuing networks or simulator models). Concepts do not map 1-on-1 to each other: system models basically construct the system (such as an architectural model), but they are not concerned with thread levels or priorities, which are important to system analysis models. They are both different

views. A model-driven approach can close this gap, because of the automation and abstraction concepts, but also because of the availability of transformation engines and domain specific modeling languages.
In order to close the gap between system models and system analysis models and to prevent terminology confusion, the following definitions have to be understood:

1. *System model* - a system model describes the structure of the entire system, including software, hardware and the allocations between them
2. *Performance annotations* - performance annotations describe specific parameters that can be attached to model elements. In this thesis, they only describe SPLICE middleware parameters, that are important for calculating the responsiveness of the system
3. *UML System analysis model* - a system model annotated with performance annotations represents a system analysis model.
4. *Calculations* - calculations can be made only on a UML system analysis model. The outcome of the calculations are various responsiveness metrics.
5. *Simulation model* - a simulation model is an UML model that describe only the concepts necessary for simulation. Usually, only behavioral flows that uses active or passive resources are expressed, as well as a deployment diagram.
6. *Formal system analysis model* - a formal system analysis model is a queuing network or Markov model. In this thesis, a queuing network is used as the underlying implementation methodology.

System engineers describe systems and structure (using a system metamodel) and add responsiveness related parameters to the models (using a performance metamodel). This results in a *UML system analysis model* and contains structural and behavioral models that describe the system (or part of the system). Those models are used as input for various *calculations* that calculate the responsiveness of a system analysis model. In order to *simulate* a system analysis model, a transformation needs to be made from a system analysis model to a simulator model. Concretely, this means that the system analysis model is made *executable*. The simulator translates the system analysis model into a formal system analysis model and runs the simulation. The simulation results are presented to the user. This "chain" is depicted in figure 3.5 on the next page. The advantage of this approach is that system engineers can compose the strucure of systems in a modeling language that they understand (UML) and that performance engineering activities can be integrated within the current system development process.

3.3 Required methods and tools

Generally, performance problems may be so severe that they impact various stages of a traditional software (or system) development process. Consider for instance the Combat Management System developed within Thales Netherlands B.V. : after the system has been built, integrators discover when using the system that it does not respond within the required timeframe that is stated in the requirements, e.g. the time that the system needs to display a vessel is not within 1000ms (1 second) but in 95% of the time within 1100ms. As a result, the integrators need to give the system back to the system engineers and automatically the questions raises what the exact problem is that caused the extra delay. If the system engineers are lucky, they can fix the problem and just need to change minor things, such as connection parameters, but usually they are not that lucky. In the worst case, they need to change the architecture of the system in order to conform to requirements. This leads to enormous problems such as not being able to fix the problem within a reasonable timeframe and worse, changing requirements that results in unhappy customers or financial consequences.

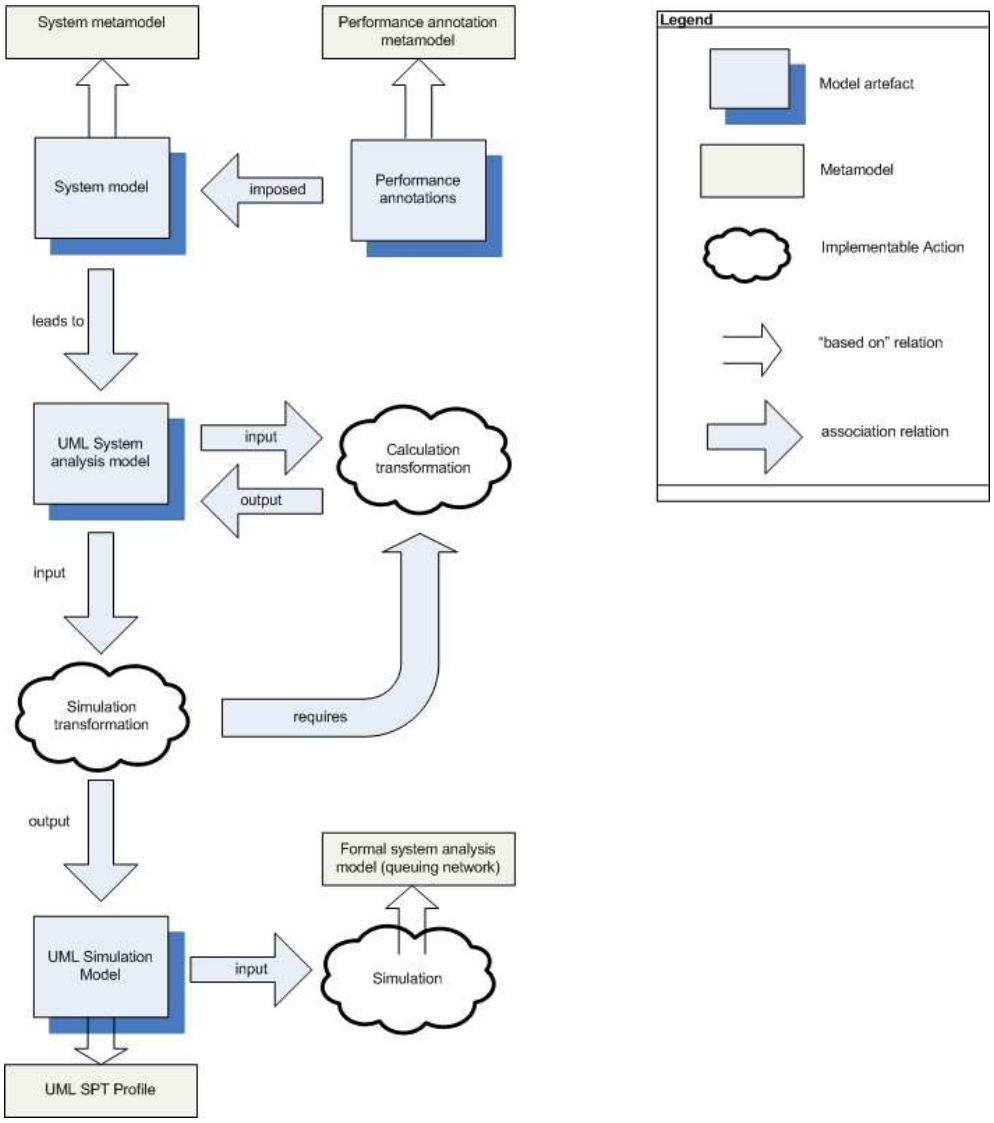

Figure 3.5: Closing the gap between system models and formal system analysis models

Thales Netherlands B.V. acknowledges this situation and explicitly expressed the need for *early performance prediction*, which should avoid the mentioned problems. Early performance analysis and prediction leads to:

- *a better understanding* of performance related problems
- *the possibility of comparing alternate systems* based on performance parameters
- *evaluation of the system* before the system has been built

The previous section already described a generic, model based approach that can be used for early performance analysis and prediction. Early performance analysis leads to a an initial understanding of the dynamic system behavior, whereas early performance prediction can be achieved by using simulations.

In [BDMIS04], software performance is stated as 'the process of predicting(at early phases of the life-cycle) and evaluating (at the end), based on performance models, whether the software system satisfies the user performance goals'. This is the "heart" of model-based performance prediction, as it tries to predict performance early. Another goal is to integrate performance prediction in the software life-cycle.
Within a performance modeling process, two features should always be present [BDMIS04]: a

software performance model coupled with system software artifacts and the evaluation of the performance model in order to obtain software performance results. This is shown in figure 3.6.

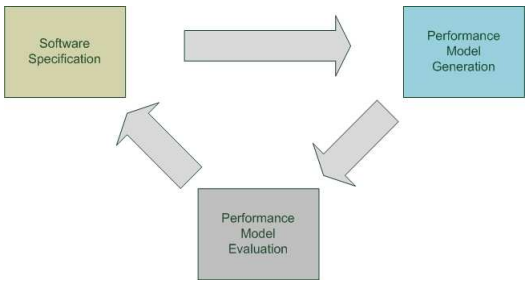

Figure 3.6: Performance modeling process overview

Software specification represent the system software artifacts that are produced during the life-cycle. Out of the software specification, a *performance model* is generated, which is used as an input for *performance model generation*. The results obtained from this step is used as feedback for the original software specification model.

The approach presented in this thesis "fits" exactly within this theoretical background: intermediate performance models are generated and by using modern UML approaches, integration of modeling activities and performance analysis can be achieved. However, there are various methodologies, both process and product based, that can be used to define exactly how the process should be executed and what design artefacts should be delivered. These methodologies are described in the next subsequent sections.

3.3.1 Software Performance Engineering

Software Performance Engineering (SPE) is an engineering process that supports early performance analysis and prediction. It is 'a systematic, quantitative approach to construct software systems that meet performance objectives' [Smi02]. Its main goal is to measure 'the degree to which a software system or component meets its objectives for timeliness' [Smi02]. This method also complies to the definition of the "Object Management Group (OMG)" [Gro05], where performance analysis is defined as 'the rate at which a system can perform its function given that it has finite resources with finite QoS characteristics' [Gro05]. Both definitions contain two important concepts: time and throughput.

Approach

SPE recognizes the fact that there exists two approaches to managing performance [Smi02]:

1. Reactive performance management - which only deals with performance problems after the system has been built. This implies an "fix-it-later" approach, which is not preferable.
2. Proactive performance management - which is the "heart" of SPE: it tries to anticipate on potential performance problems and respond to those problems on time, preferably in an early development phase.

SPE focuses on "proactive performance management" and especially on the *responsiveness* and *scalability* performance dimensions: the dimensions that could, in principle, be measured by sitting at the computer with a stopwatch in your hand.

Process

The SPE process starts with assessing possible performance risks, and identifying critical use cases (the ones most important for responsiveness). Key performance scenarios are identified

to establish performance objectives. After the performance objectives are established (usually represented as workload intensities), the performance models are constructed. This means that resource and computer requirements are added to the performance models. The next step comprises the evaluation of the performance models. If the obtained results are satisfactory, the engineer proceeds to the next performance model (e.g. system execution model) until the performance is acceptable, otherwise he needs to go back to the drawing board.
In order to apply SPE successfully, use cases and system scenarios are used in order to gain understanding of the requirements, architecture and design of a system. When all the necessary information is gathered, workloads are identified and individual processing steps are derived from scenarios. An schematic overview of the SPE process is provided in figure 3.7.

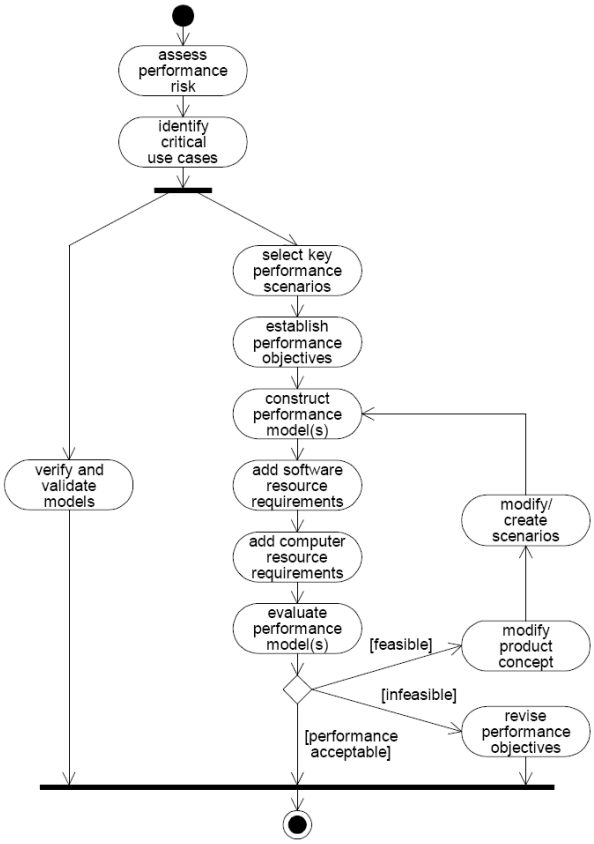

Figure 3.7: Overview of the SPE process

Models

SPE envisions that performance problems are due to inappropriate architectural choices, rather than inefficient coding. As a result, (simple) models of software processing are used that can identify problems within the current system architecture, system design or implementation plans. Therefore, modeling is deeply embedded in the SPE methodology.
Two types of models are heavily used within the SPE method: the *software execution model* and the *system execution model*. The software execution model is generally derived from UML models of the software it represents. The software execution model itself encompasses the execution behavior of the system. An example of such a software model is an execution graph, which would represent one or more workload scenarios. Solving this model provides a static analysis of the mean, best and worst case response times of a system. On the contrary, the system execution model is a dynamic model that characterizes the software performance in the presence of factors such as other workloads or multiple users that could cause contention for resources. In this model, more precise metrics are needed for calculating possible resource contention or queuing latencies. This model is typically represented as an network of queues, where every queue represents a key

resource of a system.
Within SPE, the software execution model and the system execution model are complementary models: if the obtained results of the software execution model are satisfactory, the system execution model is constructed. Otherwise the software execution model is refined until it is feasible.

3.3.2 Reasoning frameworks

Whereas SPE is concerned with models and how they should be used in conjunction with each other, a reasoning framework is a so-called "body-of-knowledge": 'it provides a 'black-box' approach for non-experts and it captures the complex theories and tools needed to be able to arrive at reliable answers about performance' [BIKM05]. Its goal is to provide a general framework for encapsulating the quality attribute knowledge needed to understand system quality behavior. Reasoning frameworks can be used within the SPE process in order to describe exactly how to reason about performance, because the models used within the SPE process are deliberately kept "simple". A general overview of a reasoning framework and its components is given in figure 3.8.

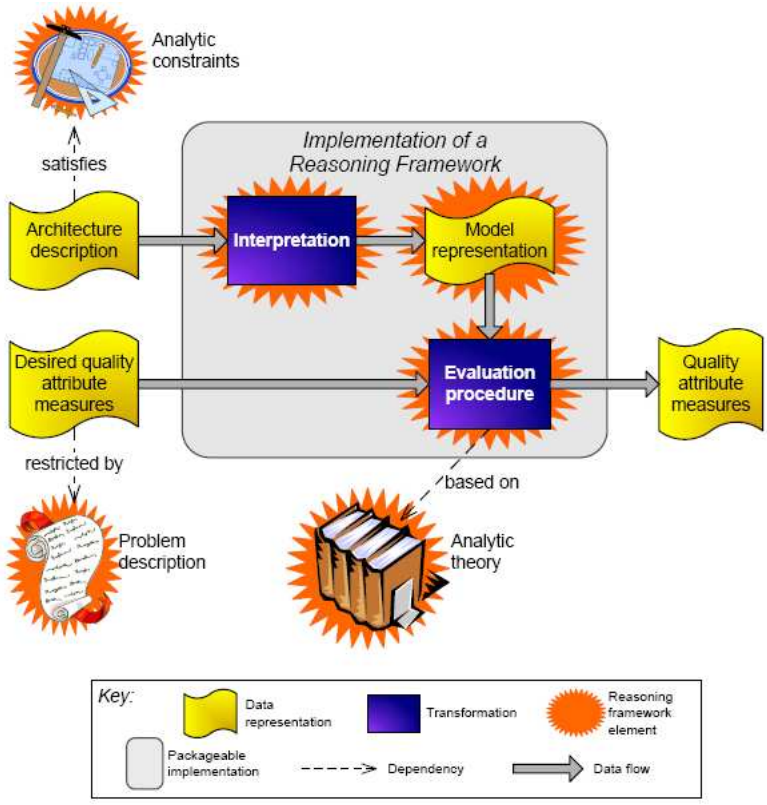

Figure 3.8: Reasoning framework overview [BIKM05]

'A system's software architecture has a significant impact on its ability to satisfy critical quality attribute requirements in areas such as performance, modifiability and reliability' [BIKM05]. A reasoning framework can be seen as a 'guide' that captures complex theories, methodologies and tools in order to assess quality attributes related to architecture. It must contain at least the following elements in order to function as a 'black-box' to non-experts:

- *Architecture description* - A description of the architecture, preferably model-based. This description must also satisfy possible analytical constraints. Reasoning frameworks use analytical theories to reason about quality attributes, usually represented as formal models.
- *Desired quality attributes* - A set of quality attribute measures.
- *Implementation of a reasoning framework* - The implementation subsystem, where the actual work is done. It takes the architectural descriptions (e.g. represented as UML models) as well

as the desired quality attributes measures as input, transforms the architectural description into a (formal) model representation, and produces output via transformations in order to predict the desired quality attribute measures by evaluating the model representation using sound analytical theories.

- *Quality attribute measures* - The output of the reasoning framework. Usually a set of measurements, but other artefacts (such as models) are also possible.

Reasoning frameworks also have their drawbacks: It is a very general framework, because it does not give any hints or guidelines of which analytical theory should be used for a specific class of problems. Another problem is that, generally, when developing an architecture for a system, the architectural descriptions (or architectural models that represent them) are usually not designed as an artefact that can satisfy analytical constraints, e.g., a queuing network. Finally, there is no feedback loop: it is not clear how obtained results affect the original model.

3.3.3 UML and its metamodeling facilities

System models and system analysis models can be developed using various methodologies, but using standard UML has a number of advantages over other methodologies. Moreover, UML will be heavily used in the future within Thales Netherlands B.V. for system modeling. It also provides good facilities for metamodeling, which increases abstraction and guarantees tool interoperability. When UML [BJR99] was invented and created, its main purpose was to make it a *general purpose* language. Before UML, there was a wide variety of object-oriented modeling languages, all trying to solve their own specific problems. At that time, there was a lot of confusion about terminology, notation issues and many other things. UML filled this gap with the introduction of a general purpose object-oriented modeling language, with a comprehensive set of modeling techniques for analysis and design, as well as structural and behavioral modeling facilities.
On the contrary, *domain specific* software engineering is concerned with modeling and analysis techniques within a specific domain. It tries to capture and define concepts that are specific towards a certain domain and provide modeling support for it. As a result, UML provides a bridge between its general purpose language and domain specific engineering which is known as *metamodel facilities*.

Metamodel architecture

Basically, a meta-model is a model of a model. This can be extended even further to a meta-meta model, which is a model of a model of a model. In this way, the number of abstraction levels that can be defined are almost infinite. The OMG [omg07] defines in most of their standards (e.g. UML) a *four-layered metamodel architecture* approach. This is depicted in figure 3.9.

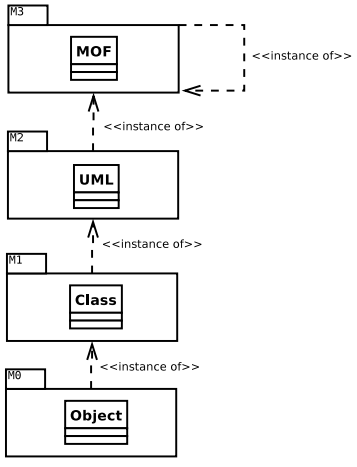

Figure 3.9: The OMG four-layered metamodel architecture

Figure 3.9 shows the four levels that the OMG defines as its architecture. It starts at the top at M3 down to M0 at the bottom. Every model of a upper layer is a metamodel for a lower situated model. So M3 is a metamodel for M2 etc. A special case is the M3 level, where the Meta Object Facility(MOF) is defined. It conforms to itself. A description of all the layers is given below.

1. The top level (M3) is where the Meta Object Facility (MOF) is defined. MOF was invented because the OMG was in need of a metamodeling architecture that could define UML. They came up with a meta-meta model that is an instantiation of itself and conforms to itself. MOF is intensively used in the MDE (model driven engineering) process, where every model is a software artefact and the ultimate goal is to automatically construct code out of models.
2. The second level (M2) is where UML is positioned. UML is an instance of MOF (meta-meta model) and therefore UML can be seen as a meta-model. UML defines all modeling concepts according to the *general purpose* vision.
3. The third level (M1) contains classes. When modeling a system or software in UML, class diagrams are an example of a commonly used model. Because classes are first-class entities in UML, a class diagram can be seen as a model and an instance of UML.
4. The fourth and last level (M0) is where the actual objects are situated. These are the concrete instances of M1 (Classes) and thus contain the actual object data.

Metamodeling approaches

UML metamodeling has a wide variety of application areas. Examples are MDE, describing ontologies or metadata modeling. But also for bridging the gap between domain specific and general purpose languages. This thesis will focus on bridging the gap between a general purpose modeling language and domain specific software performance engineering modeling.
Within UML there are three extension mechanisms defined: stereotypes, tagged values and constraints. Some of them are supported by tools (like IBM Rational Rose) and some of them are partially or not supported at all by any tool.

Stereotypes

Stereotypes are one of the three extension mechanisms of the UML. A stereotype can extend the UML metamodel in such a way that new modelelements can be defined for a specific domain, based on standard UML entities. However, four different kinds of stereotypes can be distinguished, according to [SW01]:

1. *Decorative stereotypes* - are only used to replace a symbol of a model element. They don't define specific semantics or whatsoever. It is the 'weakest' type of metamodel extension. An example could be that a class is annotated with a decorative stereotype, which contains a specific kind of symbol.
2. *Descriptive stereotypes* - introduces a new pragmatic element. In the previous example we would still talk about an annotated class, but imagine that we would like to draw a specific kind of 'hardware node'. We could annotate a class with a 'hardware' symbol (descriptive) and we provide a new pragmatic element (the hardware node). We would then talk about hardware nodes, not about classes, but in reality that hardware node still remains an instance of the original UML metamodel (the class).
3. *Restrictive stereotypes* - are new semantic elements added to the UML. They include a formal definition of syntactical and semantical constraints and they *extend* the base language. An instance of a restrictive stereotyper remains a valid instance of the stereotyped original metamodel element.
4. *Redefining stereotypes* - can replace any given metamodel element with a new one and defining a completely different set of semantics and constraints for it. Obviously, this is the 'strongest' type of metamodel extension: they can possibly alter the UML and this can result in violating UML conformance.

Tagged values

Tagged values are an extension of the UML property mechanism. With tagged values (which are basically name-value pairs) extra information can be added to an UML element. In this way, domain specific information can be added to certain elements which can be used for specific purposes, such as configuration management or code generation. A tagged value is represented as a string between brackets in a UML model.

Constraints

Constraints are used for specifying semantics or conditions that should hold for certain model elements. A common used constraint language is the Object Constraint Language (OCL). Constraints are also specified between brackets in a UML model and a typical usage scenario would be specifying load constraints in a deployment diagram.

3.3.4 UML Profile for Schedulability, Performance and Time

Over time, the "Object Management Group" [omg07] developed the UML profile for schedulability, performance and time (SPT profile). One of the intentions of this profile is 'to provide a common framework within UML that fully encompasses the diversity of various real-time concepts and notations' [Gro05]. One fundamental objective of this specification is to analyze software models in order to predict real-time concepts.
The UML-SPT does not define analysis methods: it only provides a generic domain model, where common real-time concepts (such as resources and resource usages) are shown. This model can be used to provide a generic overview of the system and can be further specialized where needed (e.g. for a specific analysis method). This is due to the so called "gap" between system models and analysis models: e.g. when performing schedulability analysis thread priority is a key concern, but these concepts are rarely seen in system models or they do not map 1-on-1.
The "base" of the SPT profile is the "resource usage" or "general resource model". This general resource model defines abtract concepts such as "resource" and "resource instance". Furthermore, the SPT profile defines a "performance modeling" component, which can be used for performance analysis of UML models. This is depicted in figure 3.10.

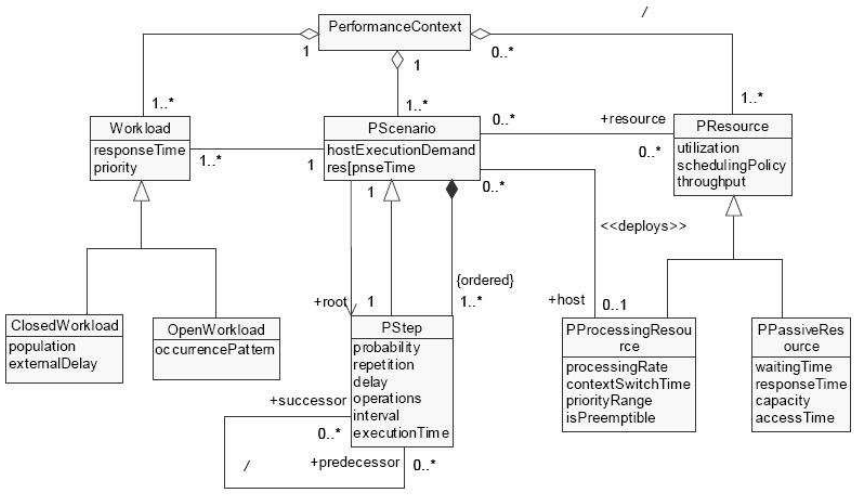

Figure 3.10: Performance analysis component of the SPT Profile [Gro05]

The performance analysis component has four goals [Hoo06]:

1. Presenting performance results computed by modeling tools or found in testing
2. Associating performance-related QoS characteristics with selected elements of a UML model

3. Specifying execution parameters that can be used by modeling tools to compute predicted performance characteristics
4. Capturing performance requirements within the design context

Previous research performed within Thales Netherlands B.V. used this profile as a means for adding performance related information to UML activity- and deployment diagrams. The result was a JAVA simulator that could interpret SPT annotated models in order to produce histograms with performance-related attributes, such as latencies, utilization and throughputs. This was useful, but it also had it limitations: the SPT profile does only support time-oriented performance characteristics and it does not solve the "gap" between analysis models and system models: the information that has to be added is still very specific and the "general resource model" of the SPT profile is too general.

3.3.5 UML Profile for Modeling and Analysis of Real-Time Embedded systems

A (recently) very new development within the "Object Management Group" is the development of MARTE, or "Modeling and Analysis of Real-Time and Embedded systems". Thales Global is also a submitter for MARTE, along with other vendors such as Alcatel and Lockheed-Martin which all recognize the need for a new approach to performance analysis of UML models. MARTE is not an addition or new version of the SPT profile, it completely replaces the SPT profile. It is also more extensive: it defines a metamodel and its underlying UML profile, it contains a domain specific language (VSL) in order to add performance expressions, full support for OCL2 and it provides detailed semantics for various performance features in order to close the gap between system models and analysis models. However, it only provides language constructs, not mappings to transform system models into analysis models and vice versa. That is something that this thesis will fill up.
However, when the MARTE draft is accepted and a final working version is released, it could be very useful for Thales Netherlands B.V. to look into this final release and investigate what could be used in the future with regards to this profile. This, because the MARTE profile looks impressive and is much more than just "another profile": it also defines semantics that describe exactly what to model and exactly how to model this. The current timescope of this project was too short to throughly investigate MARTE with regards to the current subject, but it seems promising for the future and lots of performance-modeling research is put into this final (new) standard.

3.4 Conclusion

Responsiveness and scalability are closely as quality factors: responsiveness has to satisfy user response times in a target environment; scalability means that the system can satisfy user response times in other, more demanding environments. Two important factors of responsiveness and scalability are composition and distribution and both have to be present in order to be able to design and develop responsive and scalable systems.

Concretely, Thales Netherlands B.V. defines responsiveness as the *end-to-end* latency that may occur within functional flows of the system. This relates closely to the above, more general definitions. In order to verify the responsiveness of a system, there exist (formal)analytical approaches and simulation-based methods. However, integrating formal models with system models is not preferable, as formal models are hard to understand, not all related information is available during design time and formal model concepts do not map 1-on-1 with system models. Therefore, a model-based approach is proposed that is able to close the gap between system models and system analysis models, by means of using UML metamodels in conjunction with UML modeltransformations. Intermediate models are generated with the goal of obtaining an executable simulation model, that can provide information about latencies that may occur within a system. The UML is used for expressing models that are understandable and well supported by modern tools, and for providing a high degree of automation and abstraction.

Part II

Concept development and solution-design

Chapter 4

Towards a model-based approach for early analysis and prediction of responsiveness

Previous chapters described all background information necessary for understanding the problems of Thales Netherlands B.V. and explained why early performance analysis and prediction is important. This chapter proposes a structured, model-based approach that integrates performance engineering activities with regular system development. It presents all models and transformations necessary for describing the structure of systems and presents methods for assessing the responsiveness of the composed system. Within this chapter, budget-driven performance modeling and model-driven techniques are key concepts.

4.1 Requirements

Every method or product starts with defining requirements. Based on information presented in previous chapters that described as well the current development process and general performance engineering methods and tools, a number of requirements were gathered that the final, proposed solution presented within this chapter should contain:

Requirement 1 *The proposed, changed software performance process must integrate with the system engineering process used within Thales Netherlands B.V. This means that the prototype-tool must use the UML models produced during the regular system engineering process. These models encapsulate structural information as well as behavioral information.*

This requirement is related to the current system development process used within Thales Netherlands B.V. : models that are produced during this process must be supported by the proposed approach. This in order to prevent another "standalone" solution. Integration within the development process is a key success factor for appliance in industry.

Requirement 2 *The prototype-tool has to use model-based techniques in order to close the gap between system models and formal system analysis models.*

This requirement originated from sections 3.1.3 and 3.2, that already proposed a number of models that should be present in order to transform from system models to (formal) system analysis models, with the goal of obtaining an executable model that could be simulated. This was also depicted in figure 3.5, that showed how that gap could be closed: by using intermediate models in conjunction with modeltransformations.

Requirement 3 *Performance budget information, that is currently stored within Excel spreadsheets and exchanged among subcontractors, should be used within system models in order to be able to analyse and predict responsiveness early.*

This requirement originates from the fact that currently, performance assessment is performed within Excel spreadsheets. This should not have to be a drawback by itself, but performance assessment is now performed ad-hoc. By integrating spreadsheet information with system models, system engineers are forced to assess performance at the same time when modeling the system.

Requirement 4 *Performance evaluation should be twofold: constructed system models should be evaluated using middleware related budget information and responsiveness budgets in order to assess the total responsiveness of the composed system. Verification of the obtained results is achieved by means of integrating a previously developed simulator, that is able to predict peak-loads with a certain degree of probability.*

This approach is based on the fact that the results obtained by calculating UML analysis models have to be verified. The advantage of a simulation-based approach is that it can show the overall latency of the system with a certain degree of probability. If this probability does not conform to the expected simulation results, estimated by gaining insight when using the first method, system engineers need to go back to the drawing table.

Requirement 5 *The prototype tool has to provide automation to the system engineer. Ideally, the system engineer only has to "click on a button" in order to gain insight in the performance behavior of the system.*

This requirement is based on the fact that automated tools are a key-issue for adapting and applying performance engineering within industry.

4.2 Approach

This section presents a global overview of the proposed approach that will be further used throughout this thesis. It presents an overview of all components, process-based and product-based.
One of the most important requirements of the proposed solution is requirement 1, that states that integration within the current system engineering process must be achieved. This process, depicted in figure 2.2 of chapter 2, contains one major limitation: performance assessment is performed too late (after the integration phase). Therefore, a change to this (global) process is proposed. This is depicted in figure 4.1.

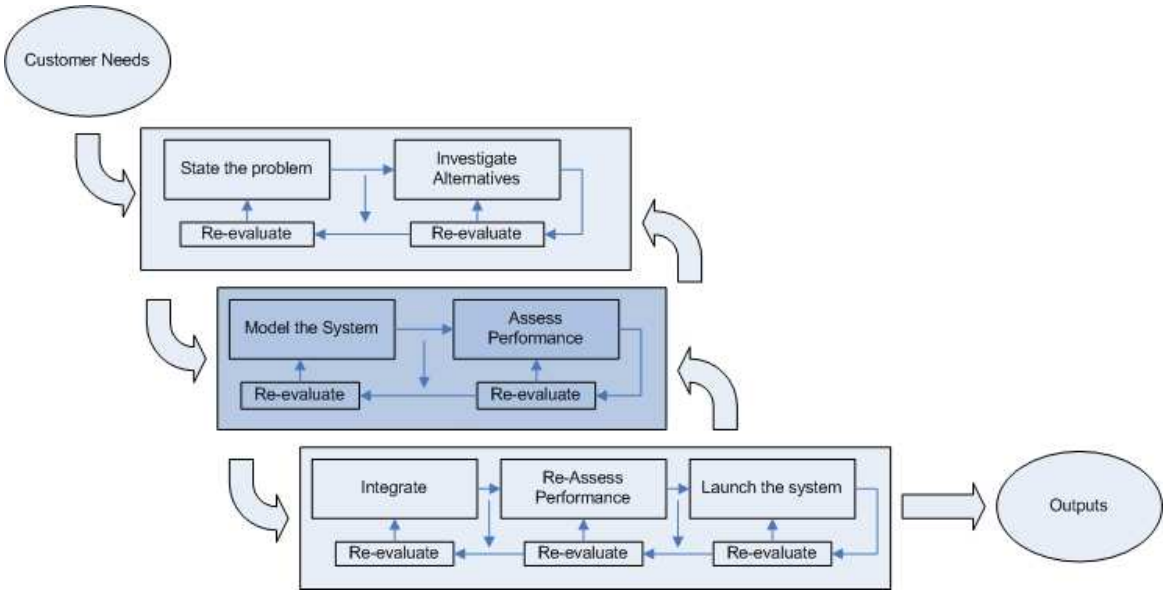

Figure 4.1: High level proposed process overview

In the new (proposed) situation, performance assessment is performed after the system modeling phase. The original performance assessment block is now named as "performance re-assessment",

with the intention that after assessing the running system, more information is gathered about the final run-time behavior of the system. As a result, this information can be used as a (updated) base for the performance assessment phase after the modeling phase: every time a runtime system is assessed, new information and metrics are obtained in order to fine-grain the analytical and simulation-based models. This is depicted in figure 4.1 on the next page.

The colored block in figure 4.1 is the central block considered within this thesis: software modeling and performance assessment are key issues here. The other blocks are not further elaborated. The separation of the three blocks depicted in figure 4.1 is introduced because many activities performed within that block are performed in parallel and usually, going back to the "drawing-board" means going back one block in time, not just one activity. This new figure emphasizes this. However, going one activity back is still possible.

In order to fulfill requirement 2, model-based techniques will be used in order to support performance engineering that makes use of regular UML models. Chapter 3 already explained the need for a model-based approach and section 3.2 presented an approach that integrated system models and UML system analysis models with the goal of obtaining an executable model (the simulation model), without directly editing formal models. This chapter elaborates further on this idea, together with chapter 5 and chapter 6.

An overview of the structure of the presented method, that is compatible with the global system engineering development process depicted in figure 4.1, is depicted in figure 4.2 on the next page. The difference with figure 3.5 of chapter 3 is that in this figure, section numbers are present. Those section numbers references to individual sections of this thesis. Furthermore, it is worth noticing that the *model artefact* "System model" is related to the "Model the system" process of the global system engineering process, whereas the *model artefacts* "Performance annotations", "UML system analysis model", "UML Simulation model" is related to the "assess performance" process. Furthermore, every "Implementable action" of figure 4.2, such as "Calculation transformation", "Simulation transformation" and "Simulation" are also part of the "assess performance" process, but they are not models, but encompasses models and actions that leads to models or other results (such as simulation results).

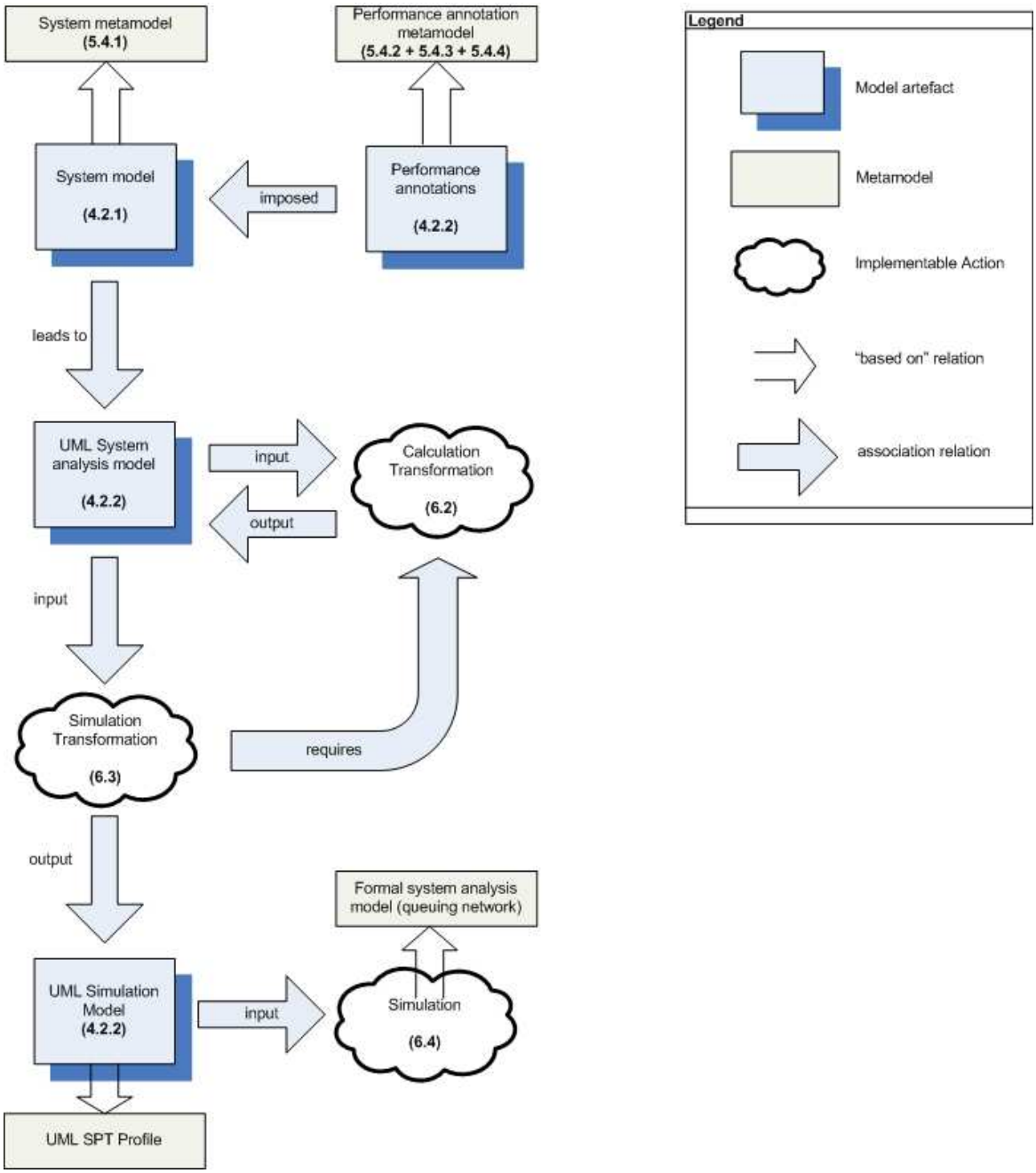

Figure 4.2: Structured overview of the proposed approach (with section numbers)

4.2.1 Model the system

One of the two activities considered within this thesis is "model the system" activity. The envisioned approach of the system engineer is as follows:

1. *Requirements gathering* - A system engineer collects the performance-related (non-functional) requirements from customers and specifications, i.e. end-to-end latency constraints
2. *Hardware modeling* - After requirements are gathered, a system is composed. The number of processing nodes are determined along with the relevant specifications (available CPU and memory)
3. *Software modeling* - software modeling consists of a number of models. The *architectural model* describes the individual CSCI's and their characteristics. The *functional model* con-

tains behavioral diagrams of individual CSCI's. The *design model* provides a detailed design, and is a refinement of the functional model. It also provides an relational model, where data-items are described.

4. *Mapping development* - mappings are models that "map" system components to software entities. An allocation diagram, that defines that 80% of a CSCI budget is mapped to a certain hardware node, is an example of such a mapping.

5. *Model constraint verification* - if certain model constraints are not met, the system engineer can't verify the performance of the system. An example of a modelconstraint is that if an entity exists in a behavorial model (such as a datastore) it should have an corresponding item in the relational model.

Figure 4.3 provides an UML activity diagram where each step in the "Model the process" activity is shown.

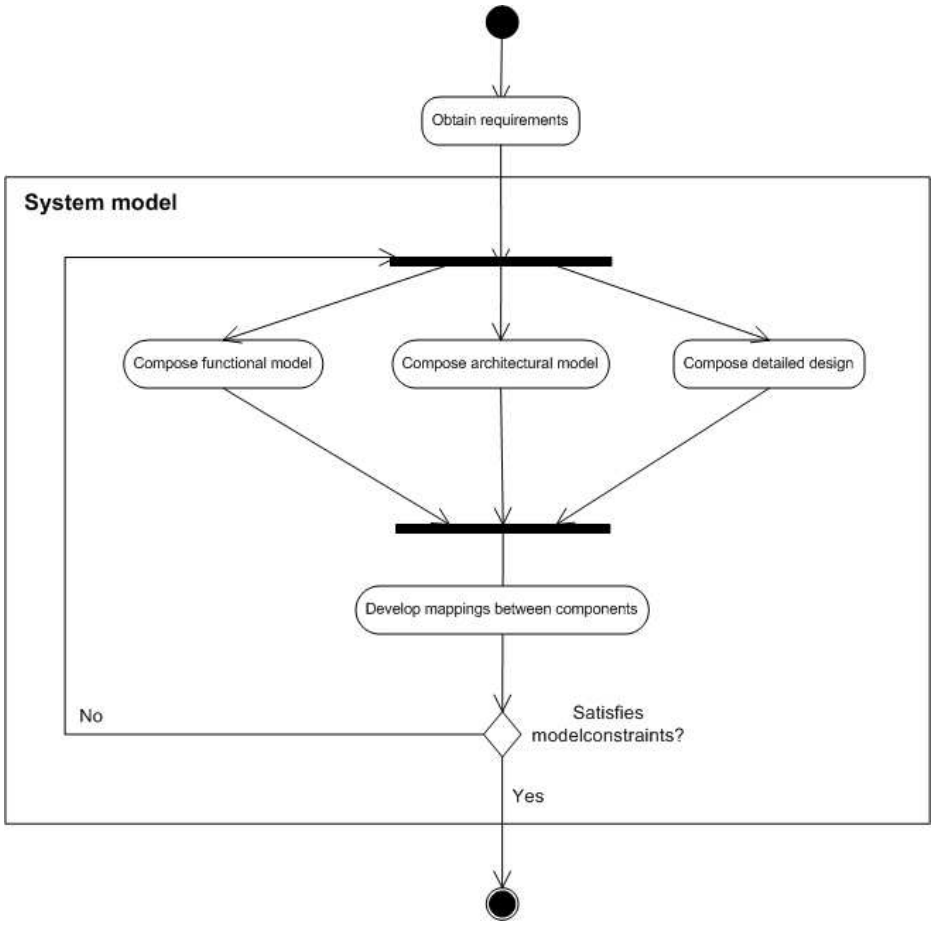

Figure 4.3: Overview of the "Model the system" process

The "model the system" activity leads to system models, depicted as a swimlane in figure 4.3. This is the same system model as depicted in figure 4.2. The goal of a system model is that performance annotations can be added and together they form the UML system analysis model.

Example system model

A system model encompasses multiple models. This section provides an overview of the most common models that together form the system model, according to the analogy described in the previous section.

The example system that will be described consist of 4 hardware nodes, which is depicted in figure 4.4.

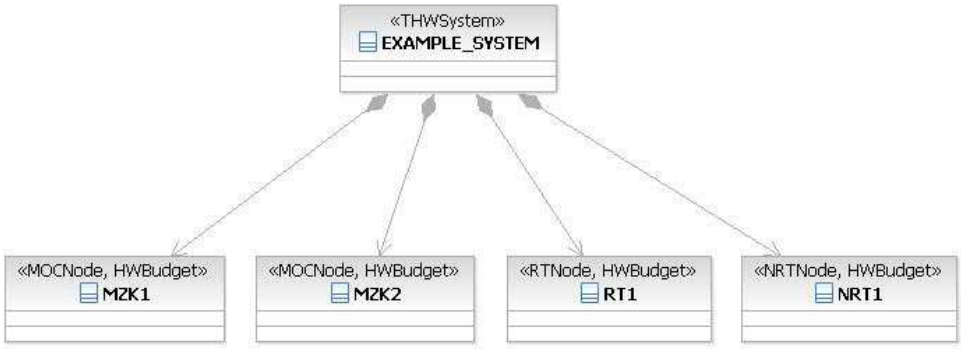

Figure 4.4: Deployment diagram of the example system

Figure 4.4 shows the deployment diagram of the example system. The top node (annotated with THWSystem) consists of two "Multi-Zweck-Consoles" (MZK's). A MZK is a hardware node with a display attached to it. Furthermore, it contains a real-time node (RT1) where most of the processing is executed. It also contains one non real-time node (NRT1), that provides various services (such as displaying historical data) to the operator.
Hardware budgets can be attached to hardware nodes: figure 4.4 shows that every hardware node is annotated with a *HWBudget* stereotype. Here, the amount of CPU and memory available for processing can be specified.

Constructing software starts specifying architecture. Within Thales Netherlands B.V. the most important part in this process is identifying the necessary CSCI's, which is a logical grouping of software components. This is depicted in figure 4.5.

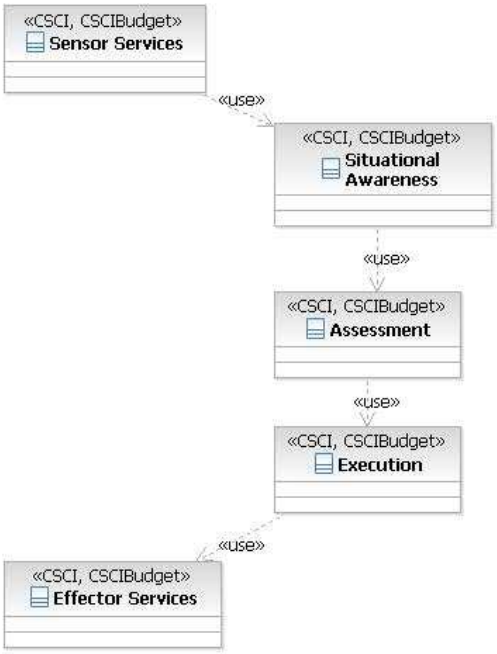

Figure 4.5: High-level architecture of the example system

Figure 4.5 shows 5 CSCI's:

1. *Sensor-services* - responsible for processing incoming sensor plots
2. *Situational-awareness* - responsible for creating tracks of sensor plots
3. *Assessment* - responsible for tracking air or surface objects, and identifying "friend or foe"
4. *Execution* - responsible for executing warfare actions (such as launching a missile)
5. *Effector services* - responsible for translating a warfare action to the actual action (such as rotating a gun)

CSCI's can also be annotated with budget information: figure 4.5 shows that every CSCI contains a *CSCIBudget* stereotype. Resource consumption, such as CPU and memory consumption can be attached. Also, middleware related performance parameters can be specified.

After the CSCI's are investigated, the system engineer can decompose the CSCI's if needed. A CSCI is the highest abstraction level of a software entity, but lower abstraction are also possible. Figure 4.6 shows this.

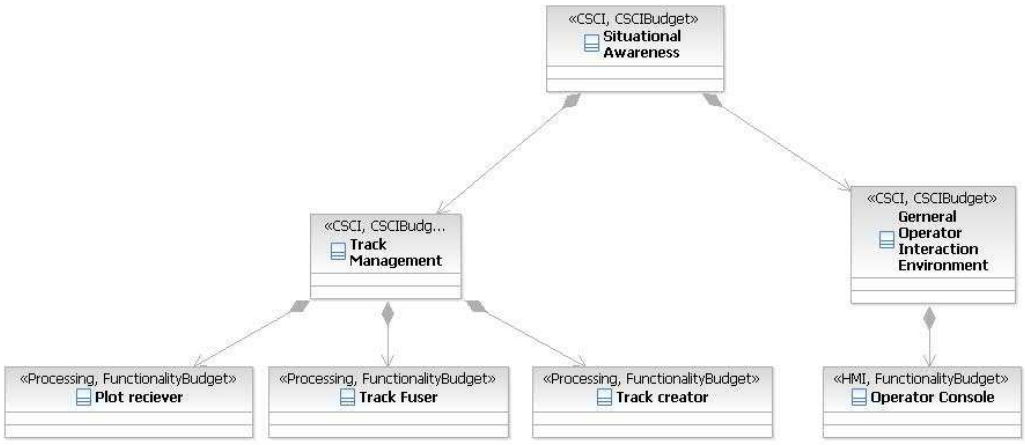

Figure 4.6: Refined architecture of the example system

The example depicted in figure 4.6 shows a decomposition of the "Situational Awareness" component. The component is decomposed using a "composite association" relation. The "Track management" subcomponent is responsible for collecting plots and generating tracks, therefore it contains three other subcomponents: "Plot Receiver", "Track Fuser" and "Track Creator". They are also annotated with different stereotypes and different budgets. The "General Operator Interaction Environment" manages the operator consoles. Furthermore, it is important to realize that CSCI's can be decomposed into other CSCI's, which indicates the usage of the 'Composite design pattern' [GHJV95].

After the system engineer collected and specified the necessary software (CSCI's) and the necessary hardware (by developing a deployment diagram) allocations are the next step in the process of specifying a complete system. Allocations determine how many resource must be allocated on a hardware node for running software entities.

Figure 4.7 shows for instance that 80% of the resource consumption of "Execution" is allocated on "MZK1" and that the other 20% is allocated on "MZK2". By changing allocation-relations, the system engineer can influence the resource consumption of the system. If the system "fits" on available resources, the total resource consumption is smaller than the available resources and the resource consumption of one hardware node does not exceed the individual resource availability, as specified via allocation relationships.

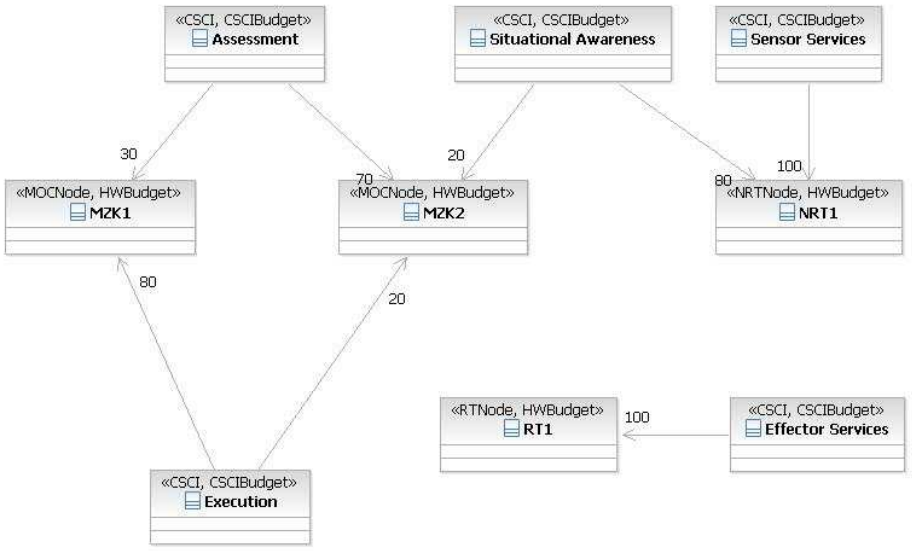

Figure 4.7: Allocation diagram of the example system

Thales Netherlands B.V. is not only interested in resource allocations by means of allocation-relationships between a CSCI and a hardware node, but also via *roles*. Figure 4.8 shows this.

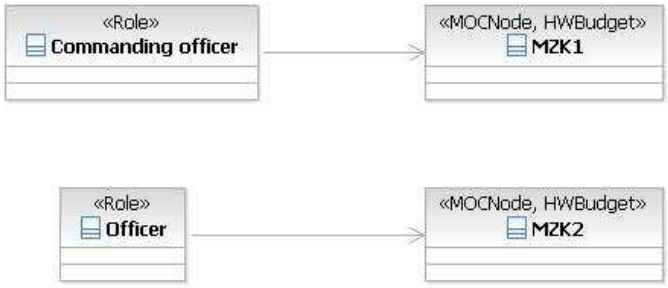

Figure 4.8: Role mapping of the example system

A role is a functional grouping of one or more multiple worksets; A workset contains one or more *executables*. Executables are classified in *processing* and *HMI*. A processing executable runs usually on a real-time node, whereas a HMI executable needs a computer with a display attached to it (a MZK), but this is not an absolute prerequisite. Finally, a role is always directly allocated on a hardware-node.

The last two mappings that are necessary for specifying a complete system are a *workset mapping* and a *executable mapping*. Workset mappings determines which "worksets", which basically is a set of tasks, belong to which roles. The workset mapping, depicted in figure 4.9 on the next page, shows two different worksets attached to separate roles. Furthermore, worksets can contain executables, which is shown by the executable mapping, depicted in figure 4.9 on the next page.

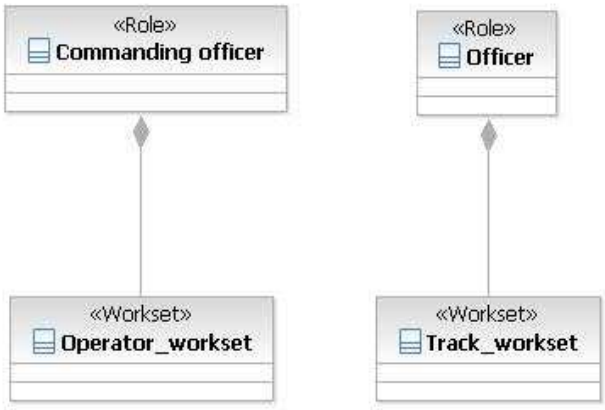

Figure 4.9: Workset mapping of the example system

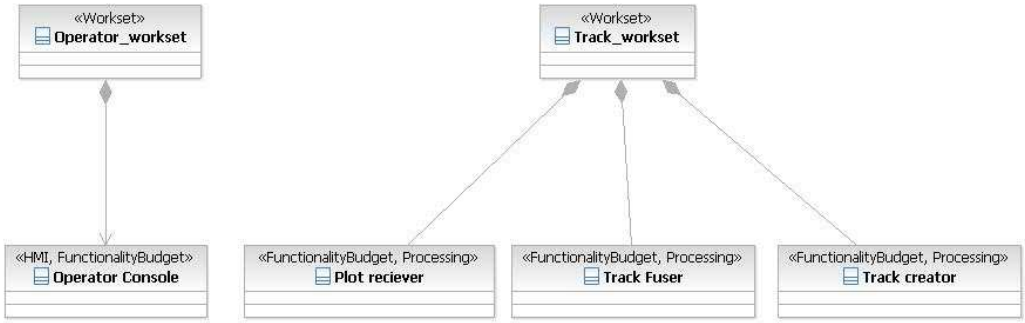

Figure 4.10: Mapping of executables of the example system

4.2.2 Assess performance

After te system has been composed, the system engineer adds performance information to the system models. The goal of this activity is the evaluation of responsiveness, which are represented by the "implementable actions" of figure 4.2. In order to be able to do so, the following steps in the process have to be performed:

1. *Annotating the models* - the system engineer annotates the functional model, architectural model and design model. The architectural model is annotated using resource budgets: the hardware model (deployment diagram) supplies resources and the software models (CSCI's) demand resource usage.

2. *Evaluate analytical constraints* - after the models are annotated, constraints are evaluated. If, for instance, a required mapping or model is missing that is a prerequisite for performance evaluation, a message is generated and the evaluation aborted.

3. *Evaluate responsiveness budgets* - after providing the required information, the evaluation process is started. The models are evaluated and if resource demand is more than provided, model-feedback is provided.

4. *Simulation* - if the budget evaluation step is satisfiable, the simulation model is generated using model-transformations. After the generation is complete, the model is fed into a discrete event simulator. A histogram of all behavioral diagrams is produced where latency information is shown. If results are not within expectation (e.g. budget evaluation is satisfiable, but somewhere seems to reside an excessive "peak-load", the system engineer needs to change the original models, until the outputs are acceptable.

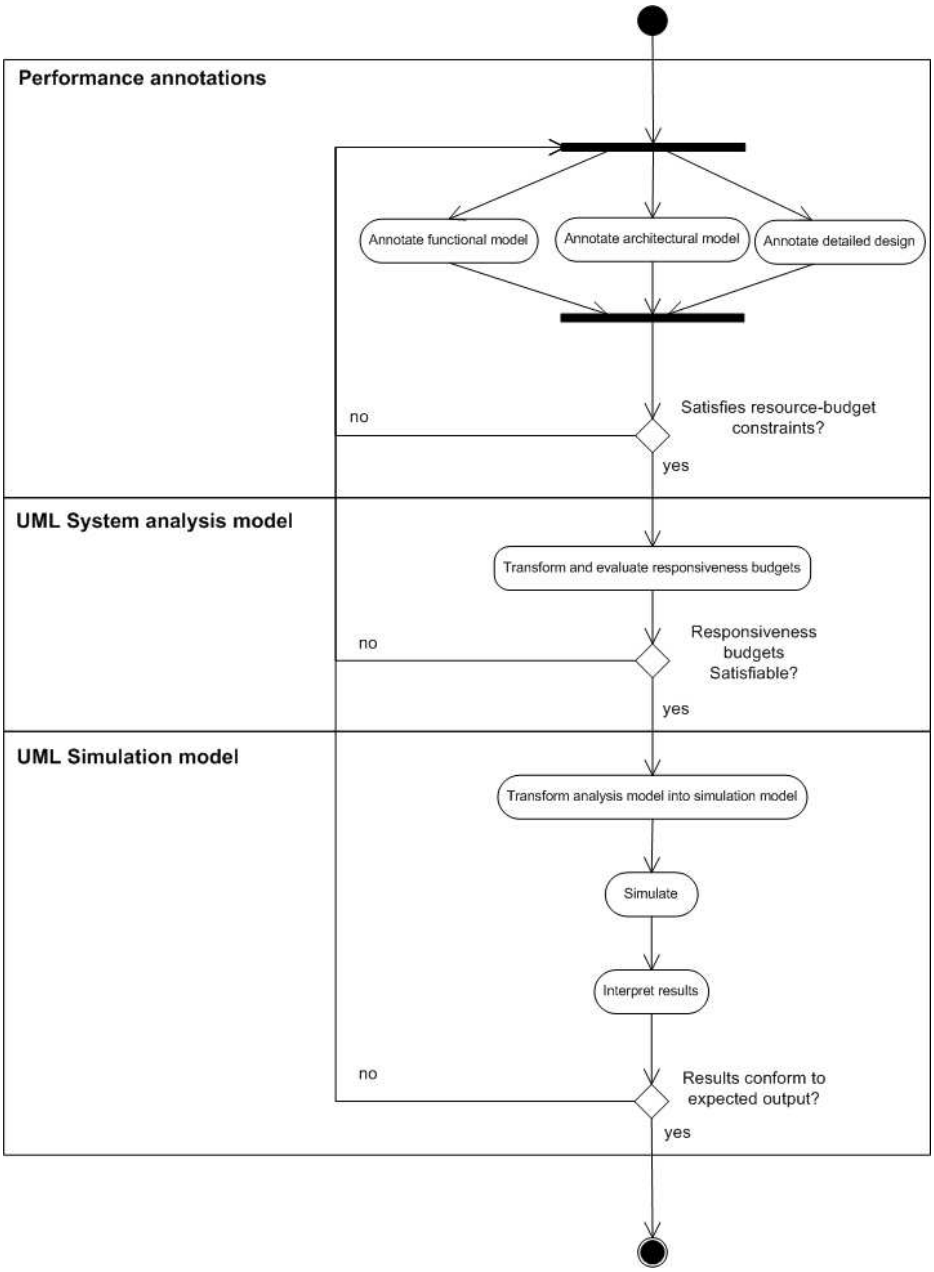

Figure 4.11: Overview of the "assess performance" process

Figure 4.11 above shows three models that are important within this process: the performance annotations, the UML system analysis model and the UML simulation model.

Performance annotations

The performance annotations are attributes or set(s) of attributes that are parameters of responsiveness. These parameters can be attached on modelelements at several levels. An example of performance annotations, attached to the architectural model, is depicted in figure 4.12 on the next page. This figure shows imposed performance annotations that are imposed on the "Sensor services" CSCI from figure 4.5. Unfortunately, Rational Rose cannot show the stereotype attributes of classes in the same diagram. However, figure 4.12 shows that at top-level, the number of processing entities (pa) is 10 and that the number of software entities that require interaction of the operator (wp, or HMI) is 12. Furthermore, both pa's and wp's have resource budgets attached, and a SPLICE budget.

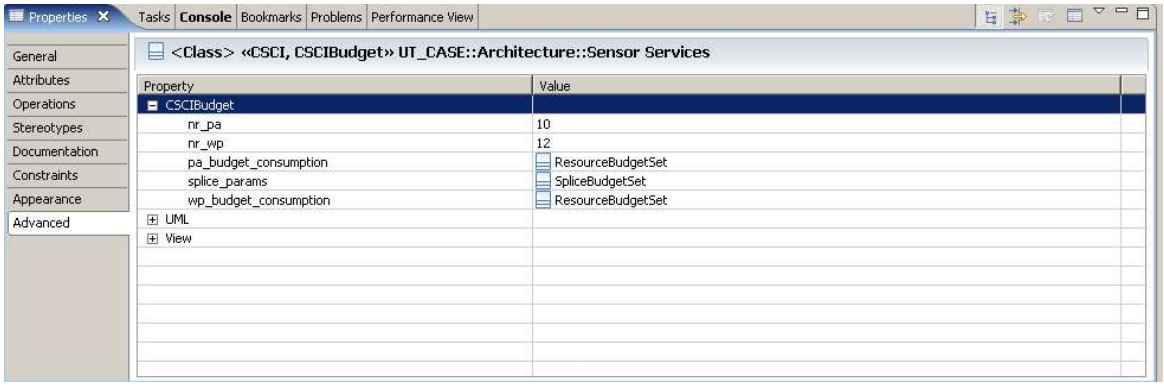

Figure 4.12: Performance annotations

UML system analysis model

The UML system analysis model is a system model annotated with performance information (at several levels). Only when system models are annotated, it can be used for analysis, by means of responsiveness calculations. When these calculations are performed, the result of the calculations are presented in the model itself, providing direct feedback to the system engineer. An example of this is depicted in figure 4.13.

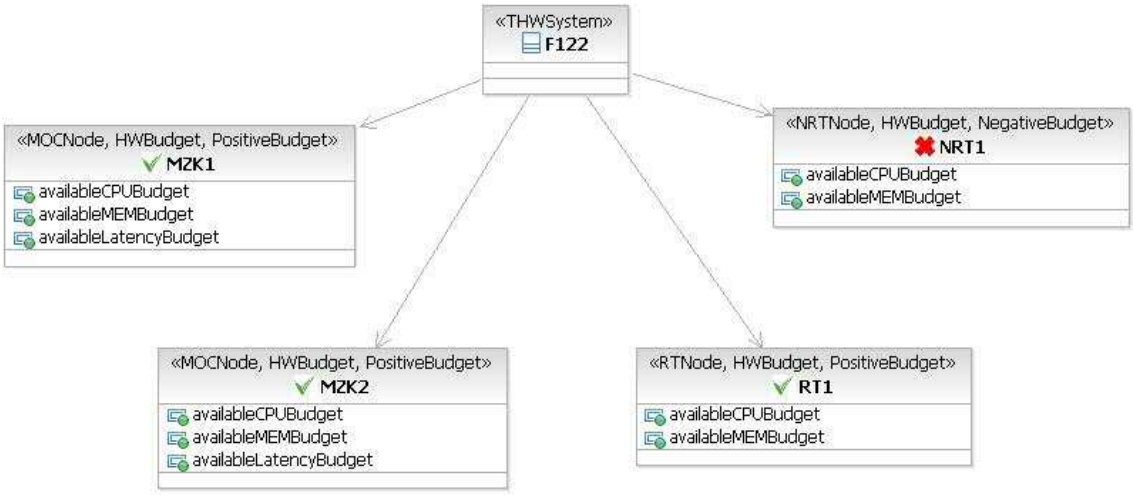

Figure 4.13: Example result of initial performance analysis

Figure 4.13 shows that MZK1 and MZK2 have sufficient budget available, whereas NRT1 has not. If the exact budget values needs to be inspected, the system engineer has to click on the corresponding attribute. Note that the complete system analysis model, as used according to the terminology presented in this thesis, encompasses all annotated system models, plus the validation information added to the model shown in figure 4.13.

UML simulation model

After the initial performance assessment (the budget verification), a simulation model can be generated out of the system analysis models. This simulation model is generated out of behavioral diagrams of the composed system. An example simulation model is depicted in figure 4.14

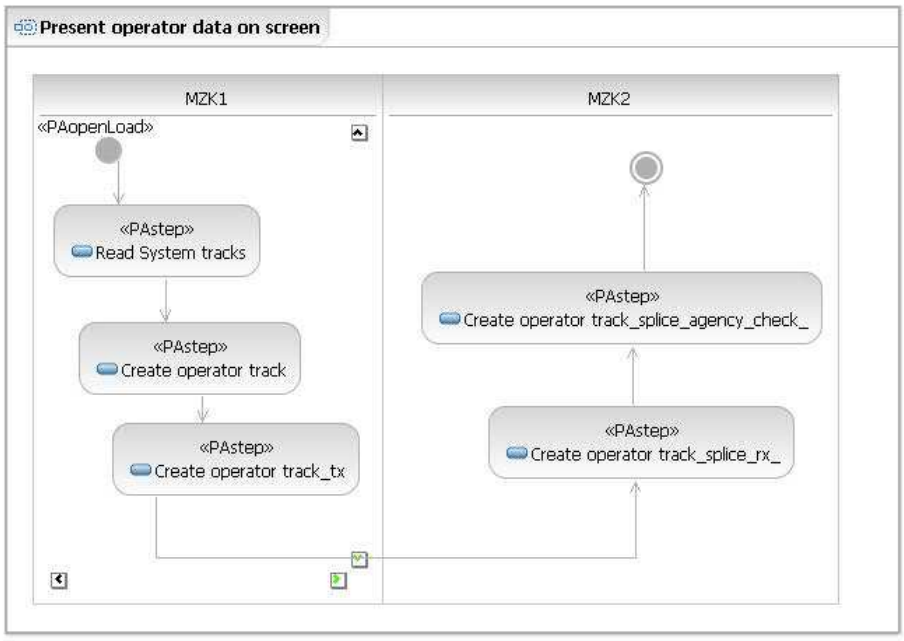

Figure 4.14: Example generated simulation model

Figure 4.14 shows an example generated simulation model. The stereotypes "PAOpenload" and "PAStep" represent workloads and the time that it takes to execute one processing step, respectively. This conforms exactly to the UML SPT Profile, previously described in section 3.3.4. The values of the individual steps are determined by the values calculated on the "UML System Analysis Models". Therefore, those calculations have to performed before the system engineer is able to transform to simulation models. The result depicted in figure 4.14 is the executable model that is fed into the discrete event simulator, and reaching this (final) model is the overall goal of the method presented within this thesis.

4.3 Supporting budget-driven performance modeling

Teh presented approach in this chapter is based on the concept of *budget-driven performance modeling*. This is a method designed within Thales Netherlands B.V. [GS07] and already applied within a number of research topics [Hoo06] [Hor06] citehor07.

Budget driven performance modeling basically means that 'the amount of cpu, memory and network processing that may be consumed to execute software to fit on available hardware, is usually specified by (non-functional)requirements' [GS07]. Expressing and tracing these non-functional requirements in artefacts produced during the software life-cycle is not easy, due to a lack of expression techniques and automated tools. Budget-driven performance modeling allows the system engineer to annotate its system model with "budgets" in order to determine resource utilization. This principle was initially developed within Thales Netherlands B.V. and further developed during this thesis. In this thesis, budget-driven performance modeling is used as a first step in the model-based performance prediction framework, because it allows specification of systems and provides the ability to verify resource constraints that may be imposed.

Basically, budget-driven performance modeling allows the system engineer to annotate system models with budget information. This was described in this chapter in previous sections. Budgets are a function of parameters such as the number of data-entities, message size and the frequency of data access. When designing a system, the budget consumers are software entities, whereas the budget producers are the available hardware resources. If a system "fits", the available budget of the consumers does not exceed the available budget of the producers. In order to determine this, the function of the budget parameters are used along with constraints. A constraint can limit a

budget usage, an example is that the available hardware budget usage may not exceed 50% in order to handle peak-load scenarios.

The budget-driven performance modeling method introduces the notion of "budgets" in order to calculate resource utilization demanded by software. By determining if a software system fits within the current hardware setup, an early performance verification step can be made. This method performs only resource utilization calculations, it does not estimate queuing delays or tries to simulate any kind of dynamic system behavior. This is an addition to budget-driven performance modeling incorporated in the framework developed during this thesis.

4.3.1 Design levels

The main idea behind budget-driven performance modeling is that regular models (artefacts) produced during a generic software development process can be used in order to calculate resource utilization. Section 2.2 described the generic development process as used within Thales Netherlands B.V. where different design levels were introduced:

1. Domain level - This is were the initial modeling takes places. Domain level modeling mainly produces functional flow diagrams were end-to-end latencies are determined. (e.g. the total time from capturing data from a radar and to present this data in track-form to the operator may not exceed 1 second). These models represent the top-level structure and top-level sequences.
2. Architectural level - The architectural level produces architectural diagrams. After the main functional flows are established, an (initial) architecture is created. This architecture encompasses class diagrams and sequence diagrams. Examples were already introduced in the previous section.
3. Design level - At this level, the low-level functional flows are created. This is the lowest level in the development process, and is basically a detailed design. Detailed architectural models are also present at this level.

Within different design levels, multiple iterations take place. Also, changes made at a higher level may impact something at a lower level. This indicates a strong hierarchy between design levels. Budget driven performance modeling supports the functional grouping of the three mentioned design levels.

4.3.2 Hierarchy and decomposition

Budget driven performance modeling leans heavily on the concepts of "hierarchy" and "decomposition". Models of the CMS contain a strong hierarchy because at top-level the detail is not very high. During a development period the knowledge and detail of the system engineers increases, so more information is added to the model in a hierarchical way. This is called "decomposition". Decomposition can occur at different design levels, structural models (e.g. UML class diagrams) can be decomposed but also behavioral diagrams (e.g. functional flow diagrams) can be decomposed. These concepts are also depicted in figure 4.15.
The hierarchy and decomposition principles are introduced with the goal of supporting the modeling of *bottlenecks* instead of modeling the complete system. System engineers will only model those parts of the system in-detail if they suspect that a degraded performance might occur. For other parts of the system, a general, global component with sufficient resource- and responsiveness budgets will be sufficient.

4.3.3 Mappings

The goal of providing mappings is that identified design levels can be linked to each other. E.g. there may exist a relation between high level flows of a conceptual model that describe the general behavior of architectural components, which resided at architectural level. Mappings enriches

the possibilities of a system engineer of describing a complete and well-defined system, at three design abstraction levels. Mappings make sure that relations between design abstraction levels are explicitly defined: they describe the corresponding vertical relations explicitly.

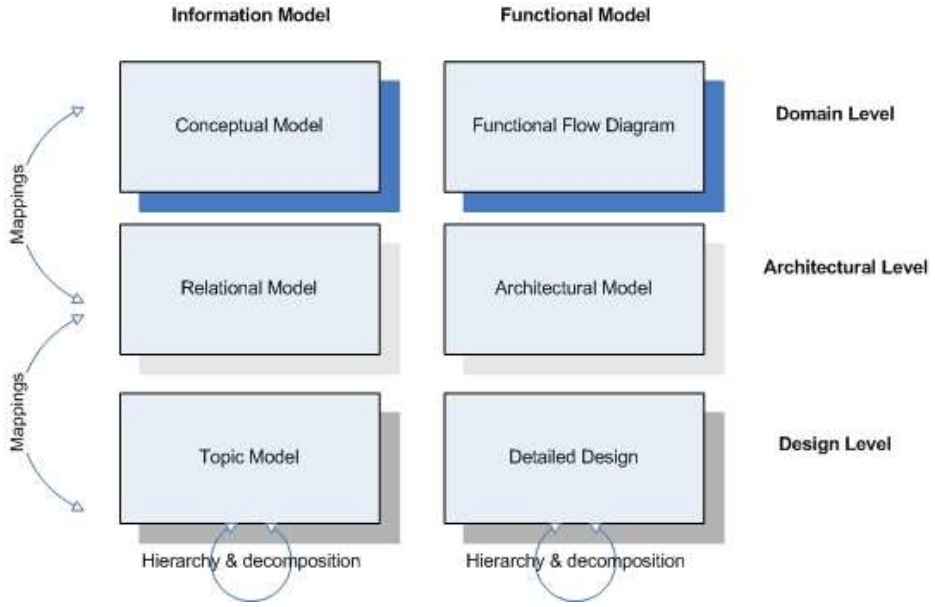

Figure 4.15: Hierarchy and decomposition in budget driven performance modeling

4.4 Conclusion

Within current research, there exists a "gap" between system models and system analysis models. By using regular models produced during the system development process and applying model-transformations, a powerful mechanism occurs that is able to reduce this gap. The presented solution proposes an architecture that supports two important activities: the first activity comprises the modeling of systems and the second activity evaluates the responsiveness of the composed system. It uses analytical-based methods to evaluate the responsiveness of the system in order to obtain an initial understanding of the expected performance behavior. Simulation is used to verify the outcome, and to show extra information. The details of the individual components presented within this chapter, will be discussed in the next subsequent chapters. Automated tool support will be provided in the form of a "proof of concept".

Chapter 5

Integrating the models

The presented approach within this thesis makes use of model-driven technologies that implements performance engineering activities. This approach relies on the use of metamodels, which are used for domain-level modeling and for describing structure. As a result, the metamodels developed within this thesis provide building blocks to system engineers.
This thesis presents two metamodels: one for describing the structure of systems and one for defining responsiveness and its parameters. It provides an overview of all individual system elements that may be used by system engineers in order to compose systems, and it describes the impact of the SPLICE middleware usage on responsiveness. Both metamodels form the groundwork for the proposed approach.

5.1 Positioning the metamodels

The previous chapter described the overall approach that was introduced in order to be able to perform early analysis and prediction of responsiveness. The next subsequent sections will describe all (meta)models in detail. The relevant models for this chapter are depicted in figure 5.1.

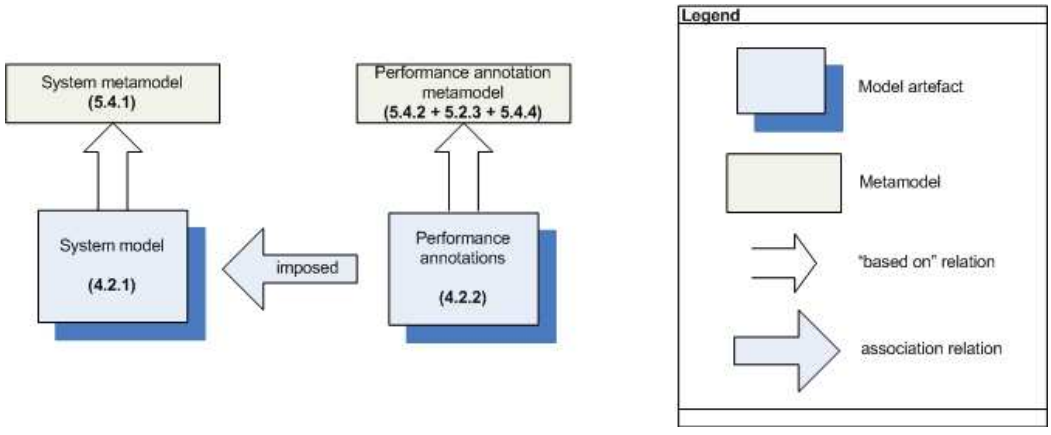

Figure 5.1: Overview of the relevant metamodels discussed within this chapter

Figure 5.1 shows two important metamodels:

1. *The System Metamodel* - which describes the structure of systems. The system metamodel is used for composing systems

2. *The Performance Annotation Metamodel* - which describes the exact performance information- and parameters that can be attached to elements of system models, based on the system metamodel

Generally, metamodels can be designed in a number of ways, something that was also described in chapter 3 in sections 3.3.3 and 3.3.3. Based on the information in those sections, and the requirements stated in section 4.1, the presented metamodels within this chapter are designed with the following objectives:

1. *It has to comply to the OMG four-layered metamodel* - because many tool vendors (such as IBM Rational and Borland Together) also use the four layered metamodel as a base for their UML implementation. Extensions of the UML also follow these directives, so in order to stay compatible among tool vendors, the proposed metamodel is also designed with the four-layered metamodel in mind. The four-layered metamodel was elaborated in section 3.3.3.
   Our proposed solution mainly comprises level M2 in the four-layered metamodel: both metamodels are expressed as UML profiles with stereotypes, which resides at M2 level. Whereas a M3 level approach was also possible (a meta-meta model) that basically leads to a MOF based language, this was not preferable because Thales Netherlands B.V. wants to model its systems in UML and therefore designing M3 level solutions will result in a solution incompatible to the UML.
   All models designed in this chapter were non-existent. By extracting information from employees, researching technical documents and consulting system documentation, the system metamodel and the quality attribute metamodel were obtained.

2. *The models are designed independently of the (possible) implementation tool* - some tools only provide sophisticated metamodeling extensions. This model is developed using state-of-the-art metamodeling techniques and possible changes due to insufficient implementation support will be dealt with in a later chapter. For instance, the UML can be extended with stereotypes based on a custom profile and MOF metamodel extensions, but not all tool vendors support MOF-level extensions.

3. *the models are designed with flexibility in mind* - That means, it has to be relatively easy to change certain parts in the future. If Thales Netherlands B.V. wants to add or change something because they have gained new understandings of their system, they should be able to do this in a simple way. Most of the time, this resulted in models which make use of certain *design patterns*, like the composite pattern[GHJV95].

5.2 A domain-level view of the structure of systems

Before discussing the developed metamodels in detail, it is important to understand a number of high-level concepts used within this thesis. These concepts show the abstract relation between system elements, constraints, calculations and quality concerns. It is not a strict *meta-meta model* according to the MOF specification, but a *domain* model that can be used for obtaining a high-level overview. This is depicted in figure 5.2.

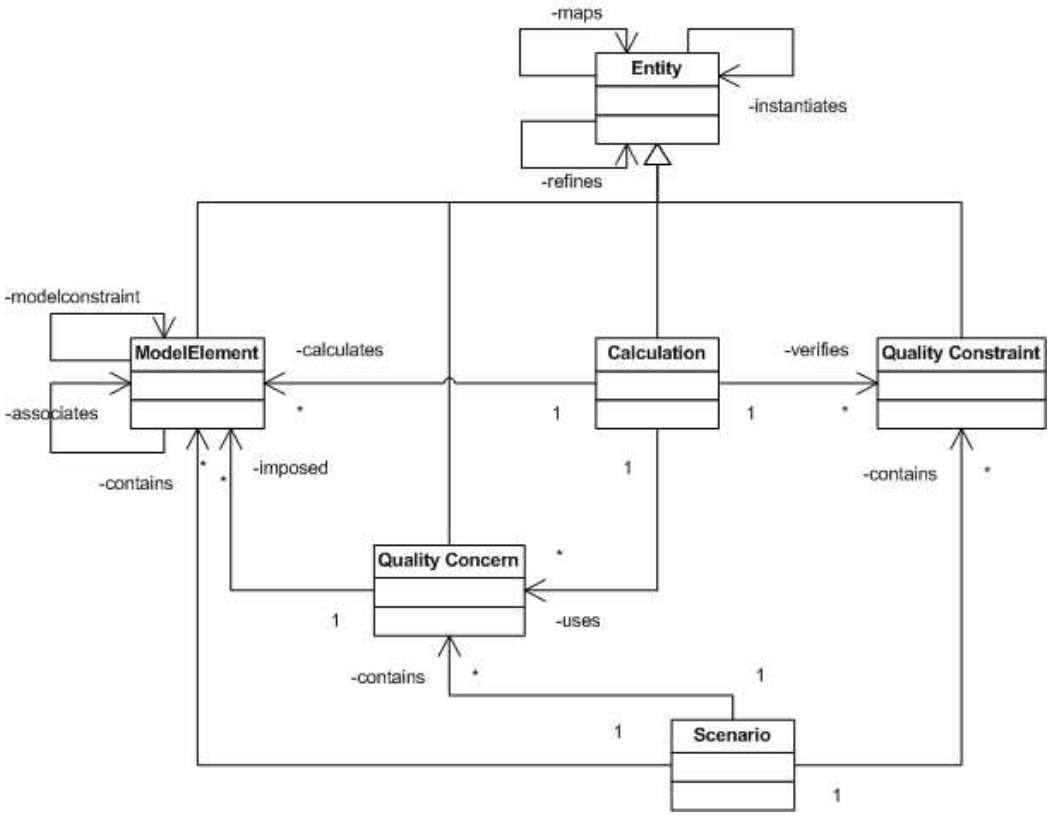

Figure 5.2: Abstract view on system structure and responsiveness

Figure 5.2 shows the following (related) concepts and entities:

- *ModelElement* - a ModelElement is an individual element that represents an item of the system. This can be a hardware-entity, or a software entity. An ModelElement can have an association with a *modelconstraint*, which means that there is a required name-based match with a ModelElement with the same name, but another type. A ModelElement can associate to itself, which means that a ModelElement can implement a composite structure. Furthermore, it is a specialization of the general *Entity* class.

- *Calculation* - a Calculation calculates the responsiveness of 1 or more ModelElements. Furthermore, it also verifies *Quality Constraints*, e.g. a loadconstraint that restrains the maximum resource usage. It uses 1 or more *Quality Concerns* that describe the exact performance parameters that a calculation have to use in order to calculate something.

- *Quality constraint* - a Quality Constraint is a constraint that can limit the outcome of a certain calculation, e.g. if a loadconstraint is imposed somewhere, this automatically means that the result of the calculation may not exceed a certain percentage. A calculation automatically verifies this.

- *Quality concern* - a Quality Concern is basically a set of *quality attributes*, which in this thesis, is represented by *performance annotations*. It is a set of quality attributes that describe the performance behavior of ModelElements.

- *Scenario* - a Scenario is a set of Quality Concerns, Quality Constraints and ModelElements. Combining these three concepts, the final goal of a scenario is to specify in a unambiguous way important scenarios, such as a combat scenario, that automatically imposes a combination of Quality Concerns and Quality Constraints on ModelElements.

- *Entity* - an Entity is the base abstract class for every entity, except Scenario. An Entity can contain *mappings*, which means it connects elements at separate abstraction levels. Furthermore, an entity can refine or instantiate itself.

5.3 A domain-level view of responsiveness

The figure with abstract domain concepts presented in the previous section shows the relation between quality concerns, calculations, quality constraints and the modelelements. This section will describe what information those quality attributes have to contain in order to assess responsiveness.
Generally, valuating if budget consumption does not exceed available resources on architectural models is not enough for assessing responsiveness. It is valuable contribution, because if the resource consumption is higher than available hardware, processing will be stalled and the system will not respond in time due to extra queuing delays. But, for assessing responsiveness other methods are needed in order to estimate the end-to-end latency of the system.

Chapter 2 stated that the CMS is a "middleware intensive" system: the usage of the middleware mainly determines the overall performance of the system. Thales Netherlands B.V. uses SPLICE as their middleware layer, so the SPLICE usage is the main factor in determining the end-to-end latency. The generic SPLICE usage is depicted in figure 5.3.

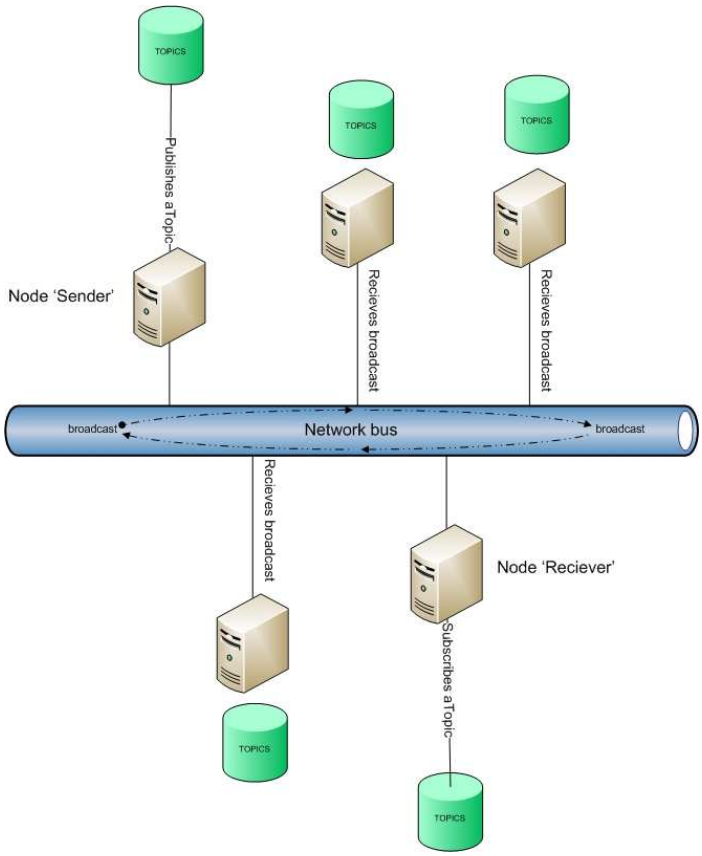

Figure 5.3: SPLICE usage overview

Figure 5.3 shows the generic SPLICE usage. SPLICE uses the publish-subscribe paradigm in order to send/receive messages. Figure 5.3 shows five nodes connected by a network. Each node contains a "topic" database. A topic is a data-item that can be sent over the network and be stored in the database, which is a "in-memory" database. The problem with this configuration is every topic gets "broadcasted" over the network. Consider for instance the situation that a "sender" node is responsible for producing a certain topic. When another node is interested in this topic, he registers with the "sender" node and every time a new update is available, the producing node broadcasts the topic on the network. So even if a node is not interested in a certain topic, he still recieves an induced network load. This important observation has led to the development of a domain model for responsiveness, which is used as the base for the AnnotationModel metamodel. The domain model is depicted in figure 5.4 on the next page.

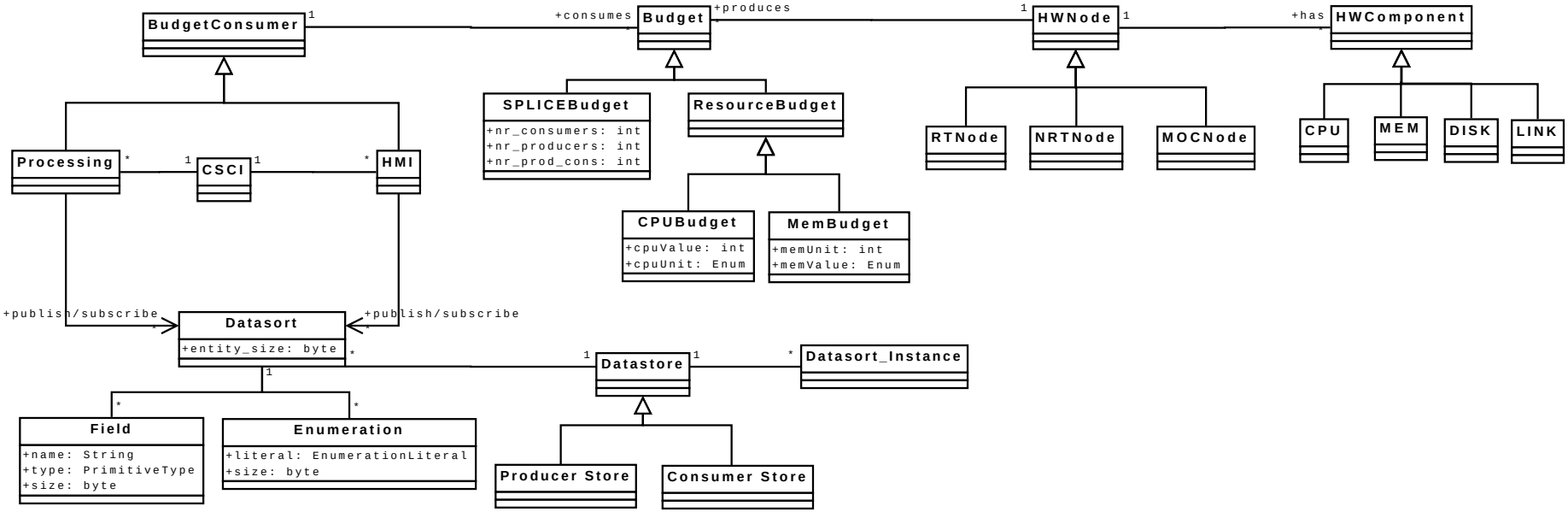

Figure 5.4: Domain level overview of SPLICE usage

Figure 5.4 shows that a *budgetconsumer* represent a CSCI (or, when decomposed, processing and hmi entities). Budgetproducers are hardware nodes providing resources. Budgetconsumers use datasorts (or topics). These are SPLICE entities that continuously are broadcasted over the network. Furthermore, datasorts are stored in a datastore.
The SPLICE agent is the piece of middleware responsible for broadcasting and receiving datasorts. Suppose that "producer A" produces a datasort that is broadcasted over the network. This leads to the following network flow:

1. the producer writes the datasort to its local producer datastore
2. the producer broadcasts the datasort over the network
3. every consumer receives this datasort, which leads to network overhead
4. the SPLICE agent of the consumer checks there is an interest in the produced datasort
5. if there is an interest in the datasort, the SPLICE agent writes the datasort in its local consumer database

Every step leads to a induced load on the hardware node where the SPLICE agent is allocated. *The induced network load produced by the producer/consumer paradigm has a major performance impact on many SPLICE based Thales systems*, and also on the CMS. In order to deal with this, behavioral models (UML activity diagrams) which are produced during the development process (functional- and design models) can be annotated with latency information, such as latency budgets and SPLICE specific parameters, in order to determine the induced load of the CMS. This determines the responsiveness of the SPLICE usage of the CMS.

5.4 The SystemModel and AnnotationModel metamodels

The SystemModel metamodel and the AnnotationModel metamodel contain those elements that are actually implemented by means of UML profiles: they are both represented as packages. Four elements from figure 5.2 of the previous section are present: Scenario, Quality Concern, Quality Constraint and ModelElement. They are represented as subpackages. This is depicted in figure 5.5 below.

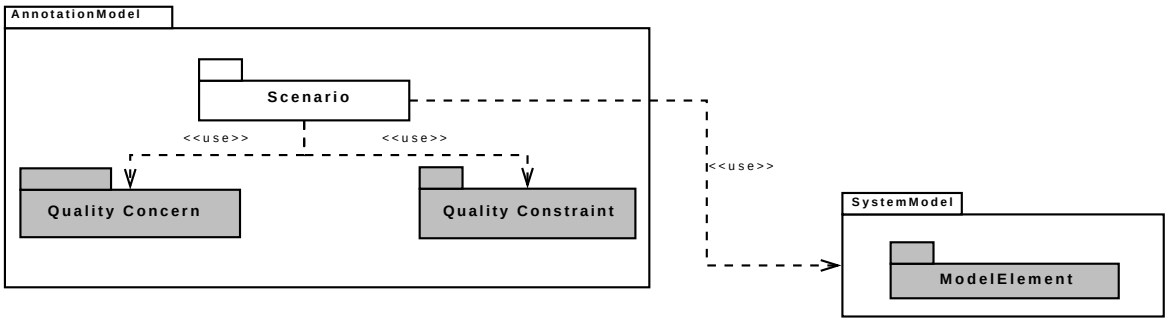

Figure 5.5: The top-level package

Three subpackages of figure 5.5 are colored grey: they are also used within the implemented prototype tool. The scenario entity is available as an UML profile, but is currently not used within the implementation. This is, also due to time constraints, a subject of future research.
The AnnotationModel comprises the Scenario package, the Quality Concern package and the Quality Constraint package. These packages contain only information that relates to quality attributes / annotations and responsiveness. The SystemModel contains only the ModelElement package, that only contains information that describes the structure of systems. A thorough description of the top-level packages is provided in table 5.1.

Package name	*Package description*
SystemModel	Base package that contains all subpackages that are necessary in order to describe the structure of systems(CMS) used within this thesis.
SystemModel::ModelElement	Contains base-elements of the system. A base-element is anything that can be modeled in order to compose a system.
AnnotationModel	Base package that contains all subpackages that are necessary in order to be able to reason about responsiveness. This package contains SPLICE related responsiveness parameters.
AnnotationModel::Quality Constraint	Contains different kinds of constraints that could be imposed on a system. A constraint adds semantical meaning: e.g. it can prevent excessive resource usage.
AnnotationModel::Scenario	Describes different kinds of scenario's that can be set that affects performance. Scenarios impact all the other elements of the QAttributeModel. They can impose constraints or set specific calculations.
AnnotationModel::Quality Concern	A quality concern is basically a quality attribute, or set of quality attributes, that describe a performance parameter of a performance calculation

Table 5.1: Description of the top-level package

Furthermore, it is important to realize that the AnnotationModel package uses the SystemModel package in order to describe exactly what kind of performance annotations can be attached to SystemModel elements. This is shown in the diagrams by a closed *extension arrow*, which denotes a (meta) extension. Also note that if elements are named the same within the same diagram they are the same.

5.4.1 The SystemModel::ModelElement package

The SystemModel::ModelElement package contains all modelelements that are needed in order to model a complete CMS. The content of the package is depicted in figure 5.6.

The dark-grey colored stereotypes / metaclasses of figure 5.6 represent the structural modelelements that add extra capabilities to existing classes. The light-grey colored stereotypes / metaclasses represent behavioral modelelements, that only adds an extra semantical meaning to existing activities.

Figure 5.6 comprises both hardware and software. Therefore, the metamodel provides *HWSystem* and *SWSystem* entities, both containing modelelements. This could either be at class level (like architectural diagrams) or on behavioral level (like behavioral flow diagrams).
The most important relations between modelelements are *mappings* and *modelconstraints*. A mapping maps one entity to another, represented in a separate diagram. An example of a mapping could be hardware-software allocation. A software component (CSCI) can be allocated to a hardware node. Other mappings are also possible, such as behavioral mappings that maps behavioral diagrams onto architectural components. Mappings are very important to system engineers, because they determine the design freedom of the system: suppose that a composed system does not conforms to its expected performance output. The system engineer cannot change the internals of the system, but he can change for instance the allocation-relationship or he can define an alternative behavioral path. The metamodel framework presented in this chapter supplies the engineers with that design freedom. In this framework, three types of mappings can exist:

1. *A refinement mapping* - refinement mappings are considered implicit mappings. This basically is a decomposition of a modelement or modelelements. An example of such a refinement mapping is depicted in the previous chapter, in figure 4.6. There, an architectural model was refined. using decomposition. The relation with the original model and the decomposition is the mapping relation.

2. *A semantic mapping* - Semantic mappings adds more detail towards a existing model. When a software component is allocated to a hardware node, the system engineer can specify how much processing time should be used for the software component. Usually, plain UML diagrams are used and the processing time is expressed in percentages. We use the cardinalities of a standard UML class diagram to express this, but the meaning of those cardinalities change compared to the original UML model. This changed meaning is the semantic mapping relation.

3. *A syntactic mapping* - Syntactical mappings allows to "work around" existing UML constructs. A system engineer can add and modify behavior of CSCI's on the fly e.g. dragging arrows between classes and behavioral diagrams. However, associations between a class and an activity violates existing UML constructs. A solution could be to represent activities as classes in a separate diagram, drag arrows between architectural components represented as classes and activities, and let some tool interprets the name-based matching between the activity and its related class. UML compatibility remains guaranteed. This construct is a syntactic mapping.

Currently, the following concrete mappings are implemented:

1. *a hardware-software allocation mapping* - that maps software onto hardware. This is an example of a semantic mapping

2. *a relational mapping* - that maps a relational model, expressed as an UML class diagram, to datastores used within behavioral diagrams. This is an example of a syntactic mapping.

3. *a behavioral mapping* - that maps activities that describe behavior to software. This is an example of a syntactic mapping.

Concrete refinement mappings make use of existing associations depicted in figure 5.6. Consider for instance the *CSCI* entity. This entity provides an implementation of the *composite dsign patter* [GHJV95] via the generalization of *SWElement* which ensures that every *CSCI* can be decomposed in another CSCI. This is a concrete example of a refinement mapping. Other (non-composite) refinement mappings are also possible via the *Role-Workset-SWElement* associations.

A *modelconstraint* is a constraint that forces some kind of convention. An example could be a name-based dependency among models or mappings. Consider the relational mapping: if a datastore exists within a behavioral diagram and we model the according relational entity in the structural relational class model, the names have to match. This match is enforced using a modelconstraint.

A description of all individual elements (colored dark-grey) of figure 5.6 is provided in table 5.2.

Element name	*Element description*
System	Abstract base class for indicating a model of a hardware or software system.
ModelElement	Abstract base class for all model elements.
StructureModelElement	Base class for all structural modelelements. These elements describe structure and are all extensions of the Metaclass Class element.
BehaviorModelement	Base class for all behavioral modelelements. These elements describe behavior.
SWElement	Abstract base-class for software elements.
HWElement	Abstract base-class for hardware elements.
OrganizationElement	Abstract base-class for organizational elements.
DataElement	Basic element for data-modeling. Can occur in relational diagrams and behavioral diagrams
CSCI	Basic software entity used within Thales Netherlands B.V.
Executable	Abstract base class for low-level executables. A CSCI can consists of multiple executables.
Processing	A processing entity is an executable that runs on a NRT or RT node. This is non-interactive.
HMI	A HMI entity is an executable that runs on a MOC node, that is, a node with a (graphical screen). This is interactive.
Role	A role is assigned to an operator (MOC) and comprises a set of authorized functions that an operator can perform.
Workset	A set of authorized functions.
HWComponent	Abstract superclass for all hardware related entities.
CPU	A processing unit.
MEM	A memory unit.
Disk	A storage unit.
Link	A physical network interface card.
HWNode	Abstract superclass for hardware nodes for specific purposes.
RTNode	A Real-Time node. Serves mostly processing entities.
NRTNode	A non real-time node. Serves mostly CSCI's components that are not mission-critical.
MOC	A node with an operator console. Serves mostly HMI entities. Requires interaction from the operator.

Table 5.2: Description of the SystemModel::ModelElement package

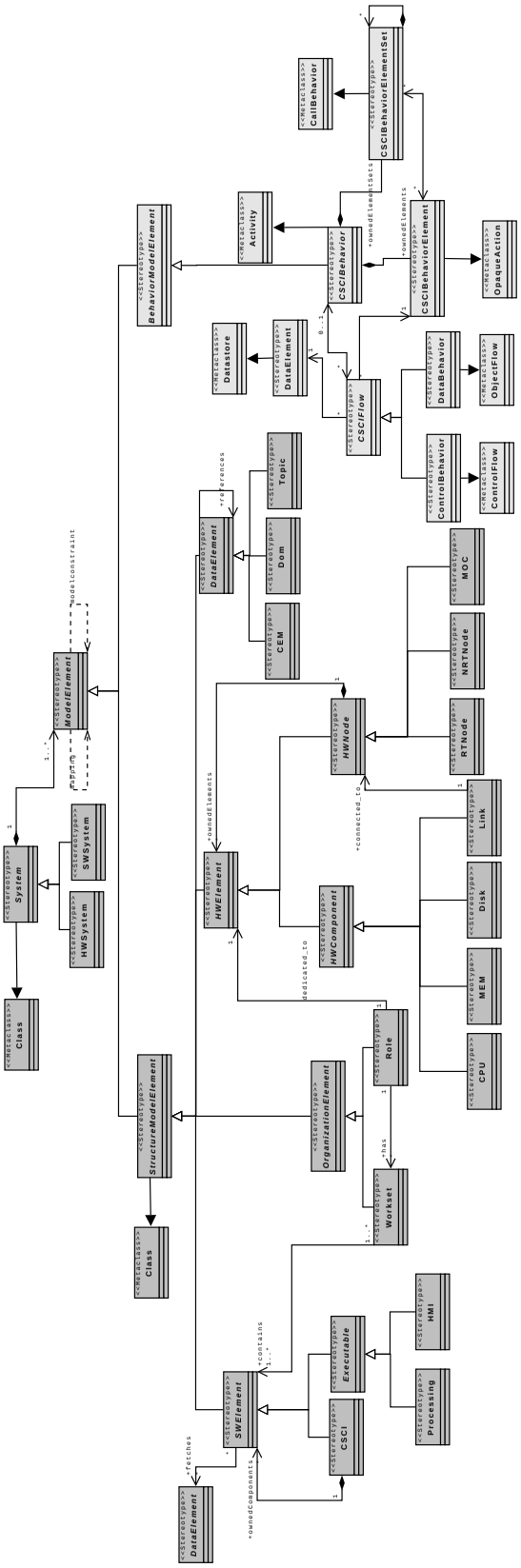

Figure 5.6: The SystemModel::ModelElement package

5.4.2 The AnnotationModel::QualityConcern package

A Quality Concern is a quality attribute or a set of quality attributes, which represent one or more performance parameters. Quality Attributes can be functional classified in groups, and those groups are represented by budgets. Examples are resource budgets or specific SPLICE middleware budgets. Every budget contains a set of quality attributes that can be attached to an element of the SystemModel::ModelElement package. Therefore, budgets can be imposed on structural elements and behavioral elements. An overview of the AnnotationModel::QualityConcern package is depicted in figure 5.7.

The most important elements are listed below in table 5.3.

Element name	*Element description*
Attribute	The abstract base class for a quality attribute.
QAttributeSet	A composite set of quality attributes. Specialized sets are defined for CSCIs, hardware, datastores and SPLICE.
QAttribute	A single quality attribute. Specializes attributes are defined for CPU, Storage, Memory and Link.
SpliceAttributeSet	Splice related quality attributes at flow level. Its members are the number of filters, the number of instances per datastore and the number of comparisons performed in a query.
DatastoreAttributeSet	Datastore characteristics. Its members are data entity size and the number of instances stored.
SpliceBudgetSet	Splice budgets at CSCI level. Its members are the number of consumers, the number of producers and the combination between them.
ResourceBudgetSet	The set of available CPU and MEM resources. Its members are available budgets.
FunctionalityBudget	Stereotype that defines available budget for Processing and Executables. Its members are budget consumption and CSCI splice related quality attributes.
CSCIBudget	Stereotype that defines available budget for CSCI's. Its members are the number of processing entities, the number of HMI entities, the corresponding budgets and Splice related quality attributes.
HWBudget	Stereotype that defines available hardware resources. Its members are resource budgets.
CPUBudgetAttribute	CPU unit and value.
StorBudgetAttribute	Storage unit and value.
MemBudgetAttribute	Memory unit and value.

Table 5.3: Description of the AnnotationModel::Quality Concern package

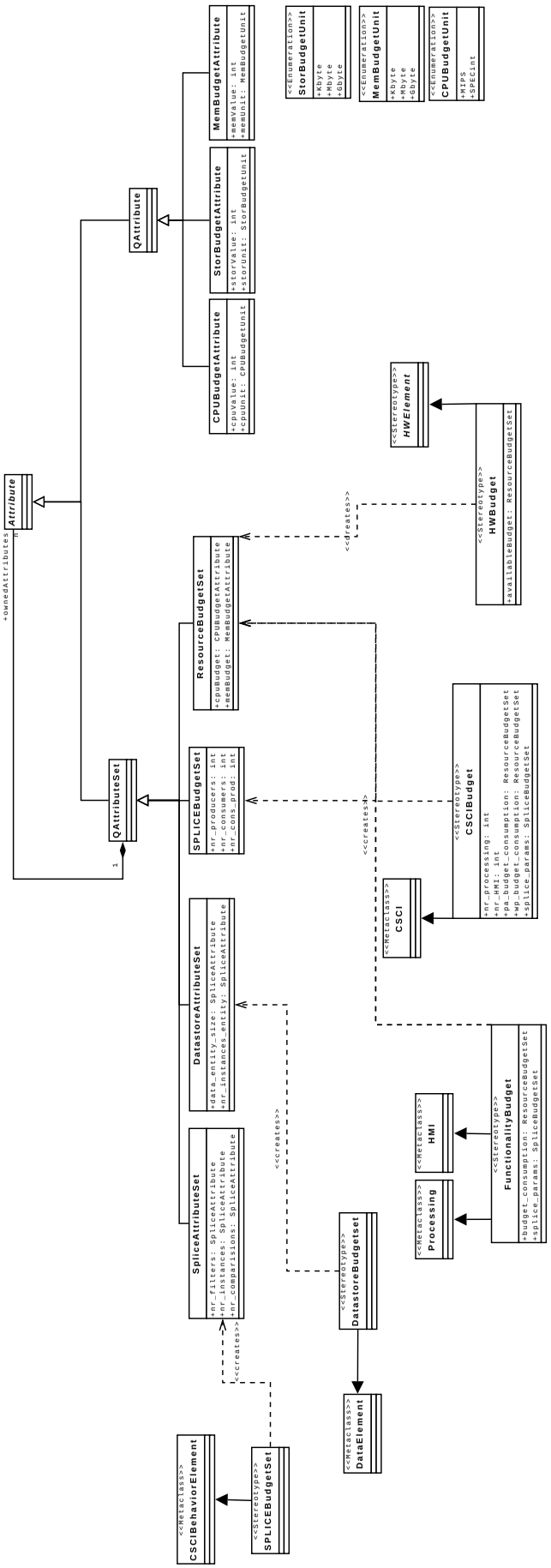

Figure 5.7: The AnnotationModel::Quality Concern package

5.4.3 The AnnotationModel::Quality Constraint package

The AnnotationModel::QualityConstraint package contains constraints that can be set at elements of the SystemModel::ModelElements package that contain budget information. A constraint can limit the resource usage that may be available for processing or can set a maximum bound on the end-to-end latency (responsiveness). An overview of the PerformanceModel::Constraint package is depicted in figure 5.8 on the next page, that shows structural constraints and behavioral constraints. Structural constraints limits the resource usage and behavioral constraints limits the responsiveness budgets. A description of all elements is provided in table 5.4.

Element name	*Element description*
Constraint	Abstract base class for any type of constraint
ResourceConstraint	Constraints specific to hardware elements. This limits available resources available for processing.
BehaviorConstraint	Constraints specific to behavioral elements
LatencyConstraint	Limits the amount of milliseconds available for a read or write process
LoadConstraint	Limits the processing frequency of a read or write processes
HWConstraint (Stereotype)	Stereotype for defining a resource constraint
LoadConstraint(Stereotype)	Stereotype for defining a load constraint
LatencyConstraint (Stereotype)	Stereotype for defining a latency constraint
FrequencyUnit	Enumeration for expressing a frequency unit
LatencyUnit	Enumeration for expressing a latency

Table 5.4: Description of the AnnotationModel::Quality Constraint package

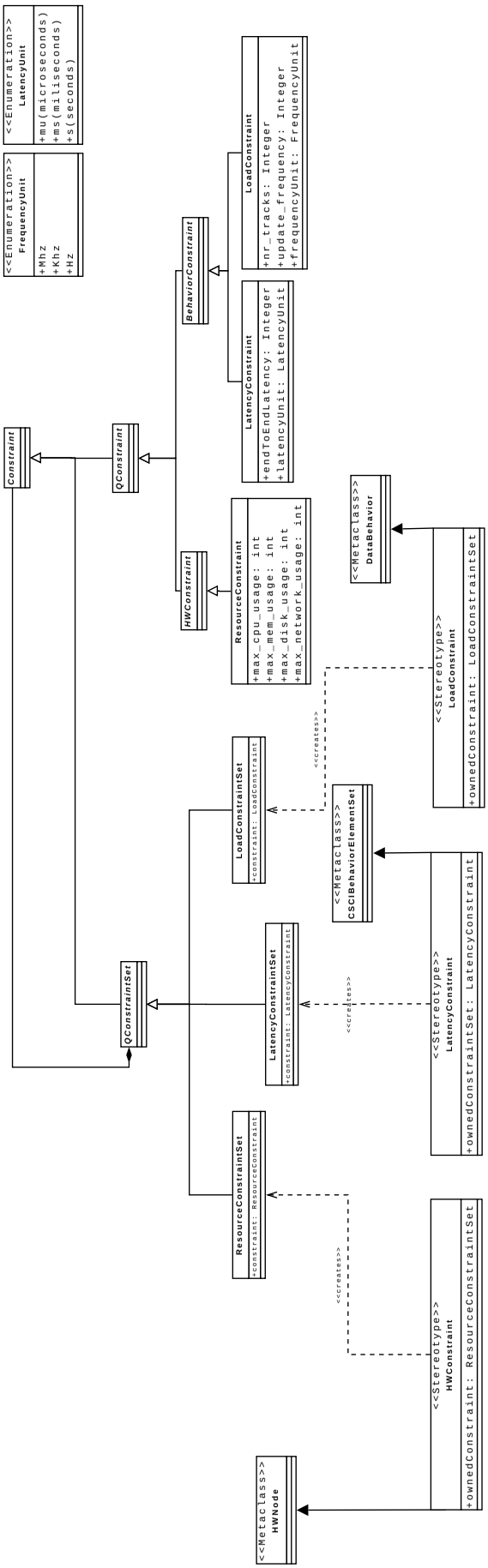

Figure 5.8: The AnnotationModel::Quality Constraint package

5.4.4 The AnnotationModel::Scenario package

Scenarios are implicitly defined by system engineers. A scenario contains a set of quality attributes and constraints. If a naval vessel is patrolling at open sea, resource and latency usage is less constrained than if it was fighting in a combat situation. Scenarios can be explicitly defined within the proposed meta-model framework, as figure 5.9 depicts.

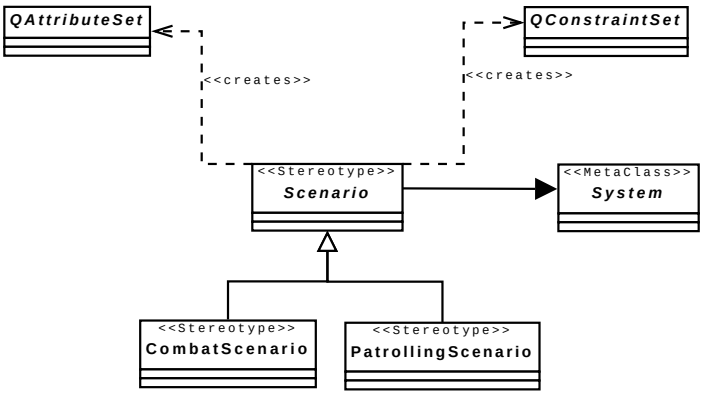

Figure 5.9: The PerformanceModel::Scenario package

Figure 5.9 shows the definition of a scenario; A scenario can be attached to a system (SWSystem of HWSystem) and comprises a set of quality attributes and constraints. These sets represents specific performance behavior, as this can vary per scenario. All individual elements are listed below in table 5.5.

Element name	*Element description*
Scenario	Abstract stereotype for scenario
QAttributeSet	The set of quality attributes to attach
QConstraintSet	The set of constraints to attach
System	Metaclass that scenario extends. A scenario can be attached to every specialization of System
Combat Scenario	Example scenario
Patrolling Scenario	Example scenario

Table 5.5: Description of the AnnotationModel::Scenario package

5.5 conclusion

In this chapter a meta-model framework was developed in order to describe the structure of systems and responsiveness related quality attributes. This led to the development of two separate meta-models: a System Metamodel and a Quality Attribute Metamodel.

The System Metamodel introduced building blocks that can be used by system engineers in order to compose systems. It introduced the notion of *mappings* which are very important, because mappings describe the allocation structure and can relate model artefacts of different design abstraction levels, with the goal of composing a responsive system structure.
The Quality Attribute Metamodel introduced the notion of resource budgets and SPLICE performance parameters. Resources are provided by hardware entities and consumed by software entities. Providing sufficient resources for consumption is a first step in the responsiveness evaluation procedure: if resource demand exceeds available resources, induced latencies will occur. However, if resource demand does not exceed available resources, it is not guaranteed that induced latencies will not occur at all. Therefore, responsiveness budgets are introduced and SPLICE performance parameters are incorporated for calculating the exact read and write latencies at functional flow level.

Chapter 6

Evaluation of responsiveness

Assessing the responsiveness of a system concretely means executing the evaluation procedure: after models have been built, mappings are defined and responsiveness parameters are attached, responsiveness can be evaluated.
The evaluation procedure is twofold: it comprises an analytical evaluation step and a simulation evaluation step. Analytical evaluation encompasses the evaluation of resource budgets (using structural models) and responsiveness budgets (using flow models). This results in a basic yes / no verdict. Furthermore, the calculated final numbers are used as input for the transformation step that transform UML system analysis models into simulation models. The executable simulation model is simulated using a discrete-even JAVA simulator and the output is compared to the analytical evaluation results. If the output conforms to the expected input (step 1), the evaluation procedure is considered successful.

6.1 Responsiveness models

The previous chapters already described responsiveness models and metamodels: it provided an overview of the required parameters and relevant modelelements. This chapter will describe exactly how responsiveness measures are calculated and what information can be obtained from the measurements. The individual sections of this chapter are the "implementable actions" from figure 4.2 of section 4.2. These are the following:

- *Calculation transformation* - calculations on system analysis models are performed by transformations on the same model. These calculations encompasses the calculation and verification of resource- and responsiveness budgets. Feedback (i.e. pass/don't pass) is provided directly to the system engineer in the same model.
- *Simulation transformation* - for obtaining an executable model, the system analysis model is transformed into simulation models. This is also performed by means of transformations.
- *Simulation* - executing the executable model and obtaining simulation results is the actual simulation. This is performed by an discrete-event JAVA simulator.

The three elements mentioned above form the base of the "assess performance" process of 4.2 and are implemented by means of "implementable actions", for clarity reasons depicted in figure 6.1, on the next page.

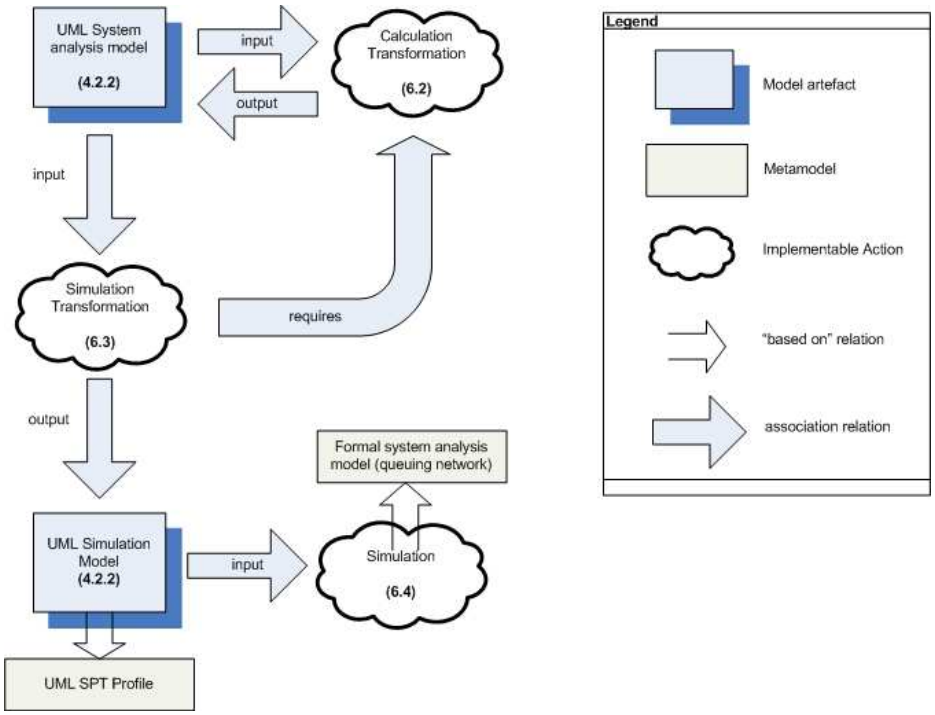

Figure 6.1: Relevant elements discussed this chapter

The "implementable actions" of figure 6.1 uses the UML system analysis model. This model contains structural modelelements, as well as behavioral modelelements. An example of a behavioral model that will be further used during this chapter is depicted in figure 6.2.

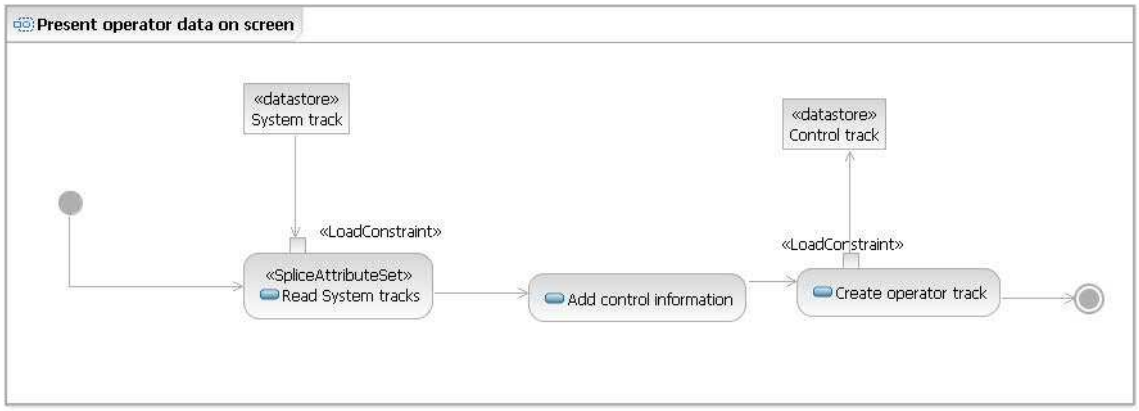

Figure 6.2: Behavioral example model

Figure 6.2 shows an UML Activity diagram that "presents operator data on screen". This is an example of a (simple) functional flow. It contains a producer, a consumer and multiple loadconstraints. The next sections will discuss how structural models and behavioral models can be used for evaluating responsiveness.

6.2 Calculation transformation

Resource budget evaluation and responsiveness budget evaluation together basically form the "analytical evaluation step" in the software performance process. This evaluation is twofold:

1. First, an UML system analysis model is obtained by annotating system models with performance annotations. As a result, resource usage and end-to-end latency can be calculated, by

using formulas and calculations from a *performance manual* [Tha07] developed within Thales Netherlands B.V. This performance manual contains information about SPLICE resource usage and SPLICE latency. This step is mainly represented by the "simulation transformation".

2. Second, a simulation model is generated. This is the "simulation transformation" step. Furthermore, the simulation model can be executed by means of simulation. This is the "simulation" step. However, the simulation model generation requires that the "calculation transformation" is executed, because its values are used as average values for distributions.

6.2.1 Evaluating resource budgets

The first substep in the "calculation transformation" is the evaluation of resource budgets: verify if the expected resource demand does not exceed available resources. System engineers can currently specify two type of resource budgets:

1. *CPU budgets* - expected SPLICE cpu usage
2. *MEM budgets* - expected SPLICE memory usage

All resource budgets are based on SPLICE usage. CPU budgets estimate the time that the system spent on SPLICE related processing and MEM budgets estimate the total SPLICE storage, that is mainly determined by the number of consumer and producer databases.
Consider figure 4.6 (refined example architecture) of chapter 4. In this example, the CSCI "Situational Awareness" was decomposed in two other CSCI's: "Track Management" and "General Operator Interaction Environment". All three CSCI's contain in this example responsiveness attributes, attached according to the structure of the "QAttributeModel" meta-model of chapter 5. For this example, the attached quality attributes of the "Situational Awareness" CSCI and their values are listed in table 6.1.

CSCI name	*Quality attributes*
Situational awareness	$nr\_pa = 10$ $nr\_wp = 12$ $pa\_budget\_consumption.cpuBudget.cpuValue = 20$ $pa\_budget\_consumption.cpuBudget.cpuUnit = MIPS$ $pa\_budget\_consumption.memBudget.memValue = 40$ $pa\_budget\_consumption.memBudget.memUnit = Mbyte$ $wp\_budget\_consumption.cpuBudget.cpuValue = 10$ $wp\_budget\_consumption.cpuBudget.cpuUnit = MIPS$ $wp\_budget\_consumption.memBudget.memValue = 30$ $wp\_budget\_consumption.memBudget.memUnit = Mbyte$ $splice\_params.nr\_producers = 10$ $splice\_params.nr\_consumers = 12$ $splice\_params.nr\_prod\_cons = 5$

Table 6.1: Description of the QAttributeModel::Constraint package

For calculating the total CPU and MEM usage of a CSCI, equations 6.1 to 6.9 are used.

$$Descriptive\,part = (4.0 + 2.0) * (nr\_producers + nr\_consumers + nr\_prod\_cons) \tag{6.1}$$

$$Consumer\,part = (6.0 + 2.0) * (nr\_consumers + nr\_prod\_cons) \tag{6.2}$$

$$Producer\,part = (6.0 + 2.0) * (nr\_producers + nr\_prod\_cons) \tag{6.3}$$

$$Memory\ usage = (Descriptive\ part + Consumer\ part + Producer\ part) \tag{6.4}$$

Equation 6.1 to 6.4 are used for calculating the SPLICE database memory storage contribution for a specific CSCI. The formulas and constant values used within these equations, are extracted from [Tha07].

$$Average\ overhead(Mbytes) = (Memory\ usage/1024) * 0.7 \tag{6.5}$$

Equation 6.5 results in an average overhead factor per megabyte of memory budget. If a memory budget of 1 Mbyte is demanded, extra memory needs to be allocated to handle the overhead, calculated in equation 6.5.

$$MEM\ usage(proc) = (nr\_proc * average\_overhead) + proc\_memory\_budget \tag{6.6}$$

$$MEM\ usage(hmi) = (nr\_hmi * average\_overhead) + hmi\_memory\_budget \tag{6.7}$$

$$CPU\ usage(proc) = proc\_cpu\_budget \tag{6.8}$$

$$CPU\ usage(hmi) = proc\_cpu\_budget \tag{6.9}$$

For calculating the total CPU and MEM contribution, the above formulas are used. Note that cpu budget is only a question of demand and supply, whereas for determining the memory budget, extra SPLICE related calculations are performed.

This example, followed by equations 6.1 to 6.9 leads to the following calculation:

$$\begin{aligned}
Descriptive\ part &= (4.0 + 2.0) * (10 + 12 + 5) && = 162.0 \\
Consumer\ part &= (6.0 + 2.0) * (12 + 5) && = 136.0 \\
Producer\ part &= (6.0 + 2.0) * (10 + 5) && = 120.0 \\
Memory\ usage &= (162.0 + 136.0 + 120.0) && = 418.0 \\
Average\ overhead(Mbytes) &= (418.0/1024) * 0.7 && \approx 0.29 \\
MEM\ usage(proc) &= (10 * 0.29) + 40 && = 42.90 \\
MEM\ usage(hmi) &= (12 * 0.29) + 30 && = 33.48 \\
CPU\ usage(proc) &= 20 && = 20 \\
CPU\ usage(hmi) &= 10 && = 10
\end{aligned}$$

After calculating the budget contribution of a CSCI, the contribution of its children are calculated. According to the "SystemModel::ModelElement" package of section 5.4.1, a CSCI can contain other CSCI's or Processing and HMI entities. The contribution of these entities are calculated according to the same formulas, with the exception that for a HMI entitie, only HMI formulas are executed (a HMI entity can't contain CSCIs or Processing entities).
After the calculations have finished, the results have to be evaluated. Concretely, this means that CPU and MEM budgets of children are summed and compared to their parent. This is performed recursively by performing a DFS(depth first search) on every software element. The pseudo-code for this algorithm is presented in the next listing:

```
void dfsBudgetCalculation(SWElement elem) {
  calculateBudget(elem);
  for each elem.child {
    dfsBudgetCalculation(elem.child);
  }
  if (hasParent(elem)) {
    if ( getParent(elem).getBudget < getParent(elem).getChildren.budgets )
      exit;
  }
}
```

The evaluation algorithm above can result in two possible outcomes:

1. *Budgets of parents are satisfiable for their children* - This means that the CPU **and** MEM budget of each parent is greater or equal than the summed budgets of their children.
2. *Budgets of parents are not satisfiable for their children* - This means that the CPU **and** MEM budget of each parent is less than the summed budgets of their children.

6.2.2 Evaluating allocations

After budgets have been calculated and child budgets are verified against their parent, allocations have to be evaluated. Typically, system engineers specify allocation relations by using mappings of the meta-model. An example of an allocation diagram was presented in section 4.2.1 in figure 4.7. Allocation relations express the percentage of the demanded budget that has to be allocated on some hardware node.
Equations 6.10 to 6.14 calculate the allocation contribution of software element components.

$$Replication\ contribution = sw\_allocation/100 \tag{6.10}$$

$$Replication\ hmi\_mem = replication\_contribution * MEM\ usage(hmi) \tag{6.11}$$

$$Replication\ hmi\_cpu = replication\_contribution * CPU\ usage(hmi) \tag{6.12}$$

$$Replication\ proc\_mem = replication\_contribution * MEM\ usage(proc) \tag{6.13}$$

$$Replication\ proc\_cpu = replication\_contribution * CPU\ usage(proc) \tag{6.14}$$

The example allocation diagram of section 4.2.1 showed that the “Situational awareness” component was allocated for 20% on “MZK2” and the remaining 80% on “NRT1”. This results in the following example calculation for “NRT1”:

$$\begin{aligned} Replication\ contribution &= 80/100 && = 0.8 \\ Replication\ hmi\_mem &= 0.8 * 33.48 && \approx 26.78 \\ Replication\ hmi\_cpu &= 0.8 * 20.0 && = 16.0 \\ Replication\ proc\_mem &= 0.8 * 42.90 && = 34.24 \\ Replication\ proc\_cpu &= 0.8 * 10.0 && = 8.0 \end{aligned}$$

However, the above example calculation is a simplified one. Actually, the top level budget of a software element is determined by the summed budgets of their children. Figure 4.6 showed that "'Situational Awareness" was decomposed in other CSCI's, Processing and HMI entities. The *dfsCalculateBudget* algorithm already checked if child budgets didn't exceed parent budgets. If they did, the algorithm terminates. If not, allocations are evaluated and parent budgets were set. In essence this means that *the top-level budget is always an upper-bound for its children.*
After allocation contributions are calculated, the demanded budgets are subtracted from the allocated hardware resource budgets. This final evaluation algorithm is showed below.

```
void dfsAllocationCalculation(SWElement elem) {
  for each (elem.child) {
    dfsAllocationCalculation(elem.child);
  }
  if (elem.isAllocated) {
    calculateAllocationContribution(elem.budget);
    demandAllocationContribution(elem.hwNode, elem.allocationContribution);
    if (exceedHwBudget(elem.hwNode) {
      markHwNode(elem.hwNode);
    }
  }
}

boolean exceedHwBudget(HWelement elem) {
  if ((elem.cpuBudget < 0) || (elem.memBudget < 0))
    return true;
  else
    return false;
}
```

6.2.3 Evaluating responsiveness budgets

After structural models are composed, annotated and evaluated, behavioral models are composed. These models describe the behavior of software components, and can be annotated with responsiveness quality attributes. These models result in an estimation of the total *end-to-end latency* of a composed flow-chain.
For this example, figure 6.2 of section 6.1 is used. Here, a simple flow model is presented: a system track is read from the corresponding datastore and a control track is produced. The parent of this process is a callbehavior action, that contains the responsiveness budget. In this case, this is 300ms (milliseconds). The read process has a "SpliceAttributeSet" attached and, according to the meta-model, contains the total number of comparisons that this process has to perform. This information can be used in equation 6.15 to 6.21, which are formulas that can calculate the SPLICE overhead.

$$splice\_agency\_checking = (2 + 1.5 * data\_entity\_size)\ \mu sec/call \tag{6.15}$$

SPLICE agency checking is the process of the SPLICE daemon that continuously checks for datasorts on the network. This process is executed at every hardware node.

$$splice\_agency\_write = 5 + (4 * nr\_consumers) + (8 * data\_entity\_size)\ \mu sec/call \tag{6.16}$$

SPLICE agency write is the contribution for writing a datasort in the local SPLICE database. Is calculated only if a node contain interested consumers for a specific topic (consumer side).

$$splice\_basicread\_access = 4 + (7 + 6 * data\_entity\_size)\ \mu sec/call \tag{6.17}$$

$$splice\_basicread\_query = (0.3 * nr\_comparisions)\ \mu sec/call \tag{6.18}$$

Equations 6.17 to 6.18 describe the contribution for a SPLICE read process.

$$splice\_basicwrite = l6 + (7 + 4 * nr\_consumers) + (10 * data\_entity\_size)\ \mu sec/call \tag{6.19}$$

SPLICE basicwrite is the local contribution for writing a datasort to the local SPLICE database (producer side).

$$splice\_network\_src = 16 + (18 * data\_entity\_size)\ \mu sec/call \tag{6.20}$$

SPLICE network overhead at source processor means the overhead that is caused by sending a datasort over the network (at src node).

$$splice\_network\_dst = 14 + (35 * data\_entity\_size)\ \mu sec/call \tag{6.21}$$

SPLICE network overhead at destination processor means the overhead that is caused by receiving a datasort over the network (at trg node).

Before being able to calculate the end-to-end latency of the figure depicted in figure 6.2, some additional information is needed. The example flow-chain is allocated on "MZK1" from the deployment diagram depicted in 4.4. This can be resolved using the *role-workset-flow* mapping specification. All calculations are calculated for each individual process and the result is subtracted from the total latency budget that is specified in flow diagrams, using the *LatencyConstraint* constraint that can be annotated on *CallBehavior* actions. These latency budgets are also allocated to hardware nodes via the same mapping. Datastore properties used within flow diagrams reside at a separate diagram: the relational model. In this example, the data entity size of "System track" is set at 2 kbyte and the number of instances stored is set at 120. 10 comparisons are executed per read and tracks are read with a frequency of 60Hz. The read/write frequency can be specified via the *LoadConstraint* constraint and the datastore properties using the *DataStoreBudget* stereotype.
The data entity size of "Operator track" is set on 10 kbyte and the number of stored instances at 10. The write process (producer) writes tracks with a frequency of 60Hz (60 writes a second) that causes the SPLICE daemon to automatically publish these tracks (datasorts) on the network. This track has also 2 remote consumers, which means that there are other behavioral flow diagrams, allocated on different hardware nodes, that reads instances from this track.
Using the above information in conjunction with figure 6.2 leads to the following example calcula-

tion:

Consumer calculations:

$$splice\_basicread\_access = (4 + (7 + 6 * 2)) * 60 = 1380\mu sec$$
$$splice\_basicread\_query = (0.3 * 10) * 60 = 180\mu sec$$

Producer calculations:

$$splice\_agency\_checking = (2 + 1.5 * 10) * 60 = 1020\mu sec$$
$$splice\_agency\_write = (5 + (4 * 2) + (8 * 10)) * 60 = 5580\mu sec$$
$$splice\_basicwrite = (l6 + (7 + 4 * 2) + (10 * 10)) * 60 = 6960\mu sec$$
$$splice\_network\_src = (16 + (18 * 10)) * 60 = 11760\mu sec$$
$$splice\_network\_dst = (14 + (35 * 10)) * 60 = 21840\mu sec$$

Src node contribution:

$$(1380 + 180 + 6960 + 11760)/1000 = 20.28msec$$

Trg node contribution:

$$(1020 + 5580 + 21840)/1000 = 28.44msec(for\ trg\ nodes\ with\ consumer)$$
$$(1020 + 21840)/1000 = 22.86msec(for\ trg\ nodes\ without\ consumer)$$

The above calculation calculates two possible outcomes; The total latency that occurs at the src node (MZK1) is in this example 20.88msec (milliseconds). The contribution for all target nodes can vary, and this depends on the fact that there could be consumers present at a target node, but this is not a prerequisite. So the final contribution for every node other than the source node can be 20.88 msec or 22.86, depending on the existence of consumers at the target node. A large system with many producers and consumers can lead to a explosion of induced splice- and network load, as this calculation is performed for every behavioral diagram.
After all calculations have been executed, the total amount of latency is added up and compared to the available latency budget. The result of this evaluation is a true or false answer. The final algorithm is presented below.

```
void evaluateFlowBudgets(Model model) {
  for each BehavorialFlowDiagram in model {
    Element initial = BehavioralFlowDiagram.getInitialNode();
    recProcessNodes(initial);
  }
  evaluateFlowBudgets(getHwNodes(model));
}

void recProcessNodes(Element elem) {
  if (elem.hasOutgoingEdge) {
    if (elem.outgoingEdge.target == datastore) {
      calculateProducerContribution(elem, elem.outgoingEdge.target);
    }
  }
  else if (elem.hasIncomingEdge) {
    if (elem.incomingEdge.source == datastore) {
      calculateConsumerContribution(elem, elem.incomingEdge.source);
    }
  }
  while (elem.outgoingEdge.target != null)
    if (notVisited(elem)) {
      visit(elem);
      Element nextNode = getNextNode(Elem);
      recProcessNodes(nextNode);
    }
  }
}
```

```
void calculateProducerContribution(Element elem, Datastore datastore) {
  collectQualityAttributes(elem, datastore);
  calculateProducerContribution(elem, datastore);
  subtractContribution(elem.getProducerSrcContribution, elem.getAllocatedHwNode)
  for each node in system {
    if (node contains consumer) {
      subtractContribution(elem.getProducerTrgConsumerContribution, node);
    }
    else {
      subtractContribution(elem.getProducerTrgNonConsumerContribution, node);
    }
  }
}

void calculateConsumerContribution(Element elem, Datastore datastore) {
  collectQualityAttributes(elem, datastore);
  calculateConsumerContribution(elem, datastore);
  subtractContribution(elem.getConsumerContribution, elem.getAllocatedHwNode);
}
```

6.3 Simulation transformation

After the system is composed and the resource- and latency budgets are evaluated, a system analysis model can be generated out of the system models. The system analysis model is used as input for the discrete-event simulator, that can show peak latencies among convoluted distributions.
In order to simulate the induced load, Thales Netherlands B.V. started a project with the goal of measuring and analyzing SPLICE based system. After various measurements were taken from real systems, formulas were defined that described the exact relation between induced network load and SPLICE usage [Tha07]. These formula's can measure responsiveness in absolute numbers. The previous section described these formulas. A drawback is, however, that these formulas only accept absolute numbers as input. Especially when estimating the number of incoming targets it is hard to establish an exact number, as this can vary over time and is never constant. Therefore, distributions are included instead of constant parameters and the constants of previous calculations are used as an average for a distribution. When this model is simulated, it results in a histogram with tabulated frequencies of peak-latencies.

6.3.1 Transformation rules

In order to generate system analysis models out of system models, a number of transformation rules are defined that describe exactly what source model element should be transformed into a target model element.
The discrete event simulator accepts two type of diagrams as input (the minimum):

1. *A deployment diagram* - a standard UML 2.0 deployment diagram
2. *A behavioral diagram* - an UML 2.0 activity diagram annotated according the UML SPT profile

Deployment diagram transformation

The first transformation comprises the generation of a valid UML 2.0 deployment diagram out of the system models. This is relatively a straightforward task, as standard class diagrams are used to express available hardware resources. The transformation is depicted in figure 6.3.

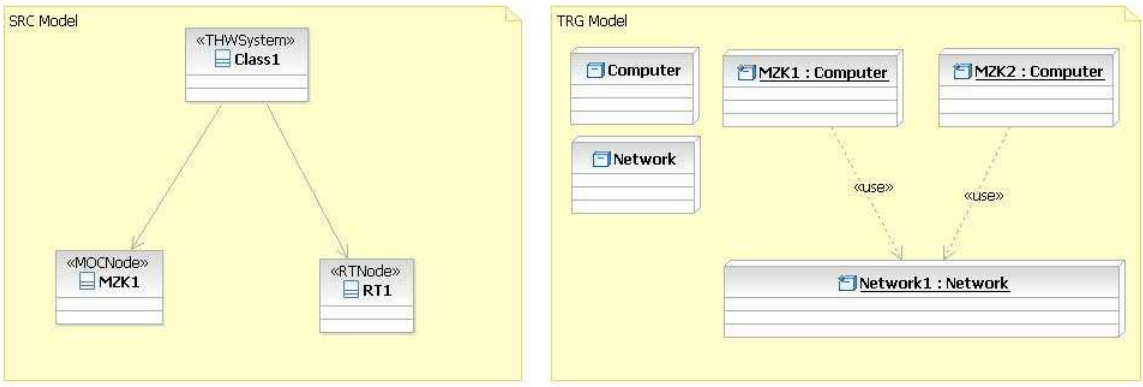

Figure 6.3: Deployment diagram transformation

The "SRC Model" box shows an example hardware configuration that could be composed. The "TRG Model" box shows the result of the transformation:

1. Collect all classes that contain annotations that are a specialization of *HWNode* (e.g. MOCNode and RTNode)

2. Create two "nodes" in the target model and name them "computer" and "network". A node is a deployment diagram entity

3. For all collected classes at step 1: create "node instances" that are an instance of "computer" and copy the names

4. Create a "node instance" of the "network" node

5. Connect all node instances of "Computer" to "Network".

After the transformation is successful, a valid deployment diagram is generated.

consumer transformation

The second (and third) transformation is the generation of valid activity diagrams annotated according to the UML SPT profile. The behavioral diagrams that describe CSCI or executable behavior are used as input. This transformation is depicted in figure 6.4.

The first "swimlane" depicted in figure 6.4 is the example source model and the second swimlane, "TRG Model" represents the model after the transformation. The following steps are executed in order to transform from the source model to the destination model:

1. Collect all consumers. A consumer is an "OpaqueAction", with an incoming arrow from a datastore. This implies a read process.

2. For each consumer, delete the arrow and datastore. Annotate the "OpaqueAction" with the "PAStep" stereotype. A PAStep represents an active service request form an active resource (such as the CPU). In essence this represents the duration of the read request.

3. If there is an initial node specified, annotate that load with an "PAOpenLoad" stereotype. An PAOpenLoad represents a workload: e.g. a workload of 200ms means that this workload is scheduled five times per second.

The consumer transformation is executed for each consumer found within an activity diagram, so the number of transformations performed can vary.

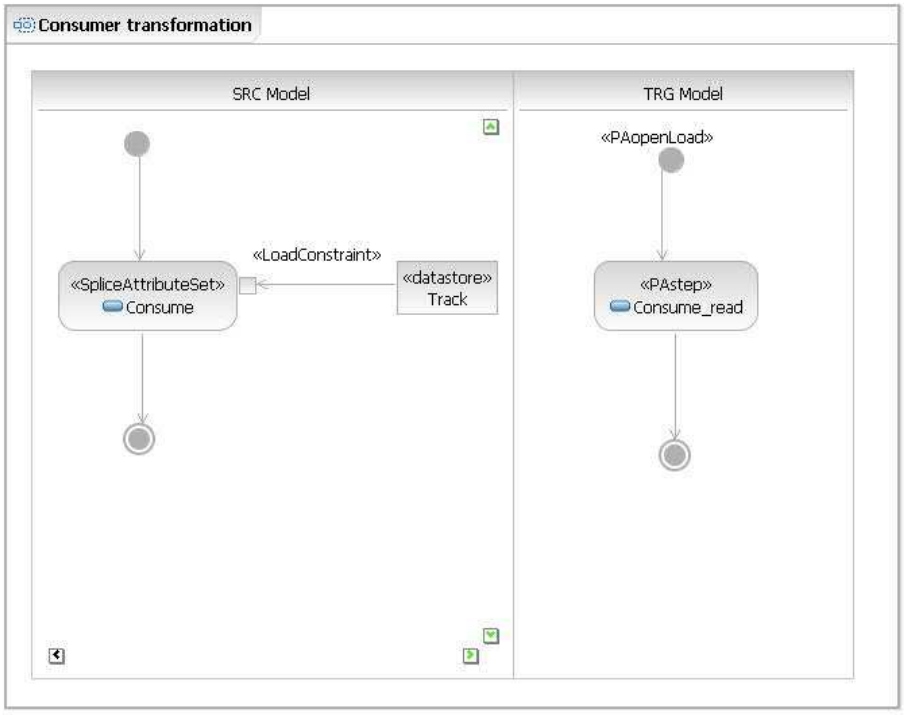

Figure 6.4: Consumer transformation

producer transformation

The producer transformation is the most complex transformation. This is due to the fact that induced loads have to be simulated. A producer that broadcasts tracks causes an induced loads on all other nodes and the induced load may vary upon the fact if a node contains interested consumers. This concept is depicted in figure 6.5.

Figure 6.5 encompasses the following transformation steps:

1. Collect all producers. A producer is an "OpaqueAction" with an outgoing arrow towards a datastore

2. For every producer, create a "PAStep" for the local SPLICE write contribution and network load at src node (TRG Model)

3. For every other node (TRG Model') create "PASteps" for network overhead and SPLICE agency checking

4. If, at TRG Model', the SRC Model datastore is read, add a local SPLICE consumer write "PAStep".

5. If there is an initial node specified, annotate that load with an "PAOpenLoad" stereotype.

Figure 6.5 is not a trivial one. For determining the TRG Model node, system model mappings are used (behavioral diagrams are mapped to worksets, worksets to roles and roles to a hardware node). TRG Model' is also a dynamic generated swimlane, as it depends on how many nodes the composed system contain. At every node, datastores needs to be checked in order to obtain interested consumers: in this case the datastore "Track". If so, then an interested consumer node receives an extra SPLICE consumer write contribution. The whole chain represents the SPLICE load for one producer only. For multiple producers, the model becomes easily very complex.

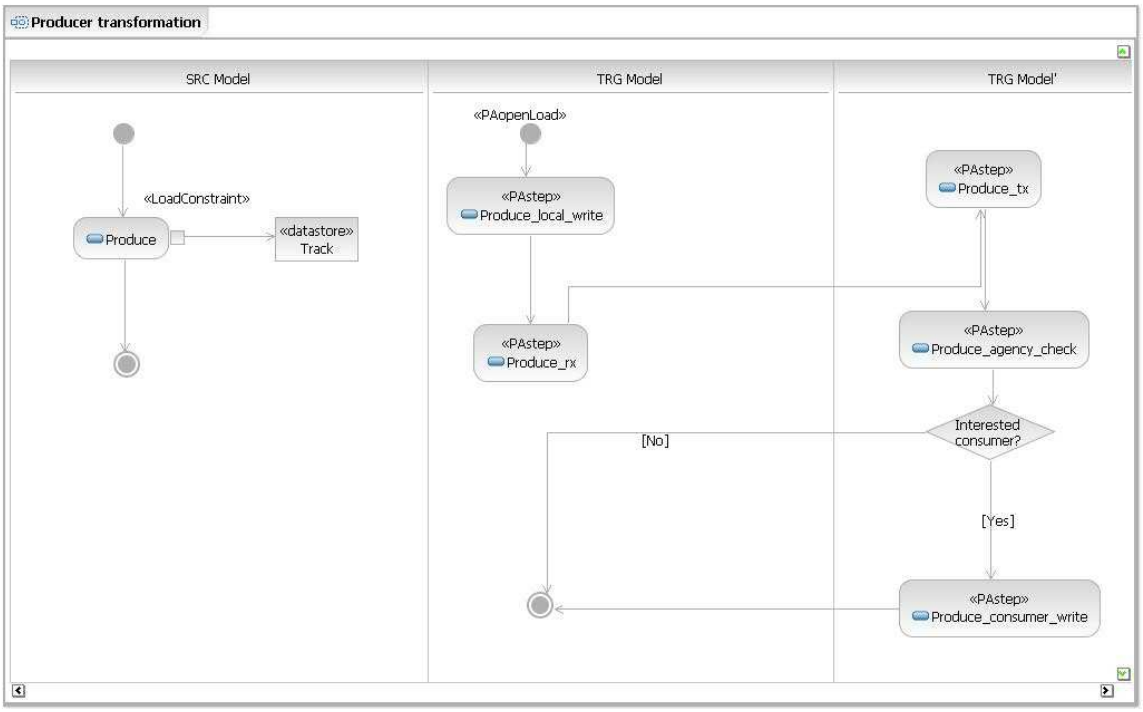

Figure 6.5: Producer transformation

6.4 Simulation

After the analytical model is constructed and calculated, simulation models are generated. These simulation models are transformed using the transformation rules presented in section 6.3 and use the UML SPT Profile [Gro05]. The main purpose of this simulation is:

1. *Verification of analytical models* - to check if the calculated results are "trustworthy" results
2. *Estimation of factors that cannot be shown in an analytical model* - peak loads can be the result of correlated distributions. These factors cannot be shown in analytical models.

When simulation models are generated, analytical results are used as input, and only for behavioral flow diagrams. Those are the only diagrams that can be simulated. When a producer or consumer is found in a behavioral flow diagram and the contributions are calculated, these numbers are used as input for the generation of the simulation models. These models contain processes annotated with *PAStep* from the SPT model, which represents an active service request. On that service request, a distribution (or constant values) can be attached. After that, the simulation is started.

6.4.1 Simulating system analysis models

Distributions are incorporated instead of constants numbers for each individual PAStep. Currently, the following distributions are supported within the evaluation procedure:

1. normal distribution
2. exponential distribution
3. uniform distribution
4. constant (not a distribution, rather a constant number)

The normal distribution, the exponential distribution and the uniform distribution are depicted in figure 6.6 below.

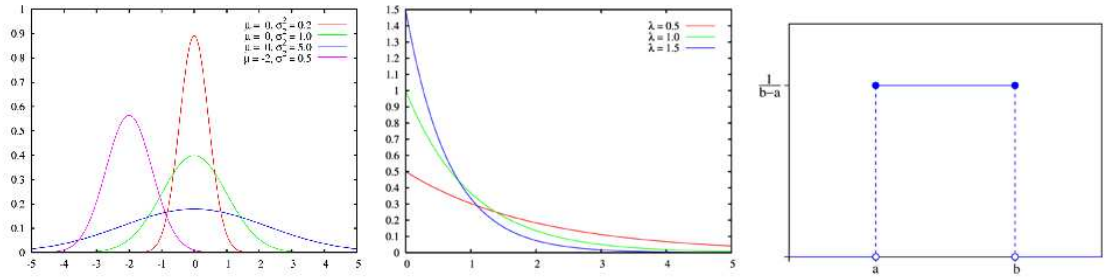

Figure 6.6: Implemented distributions

To show the impact of using different distributions within a simulation, the previous analytical example will be used as input for a number of simulation runs. For the first run, only constant values are used. This leads to the result depicted in 6.7 below.

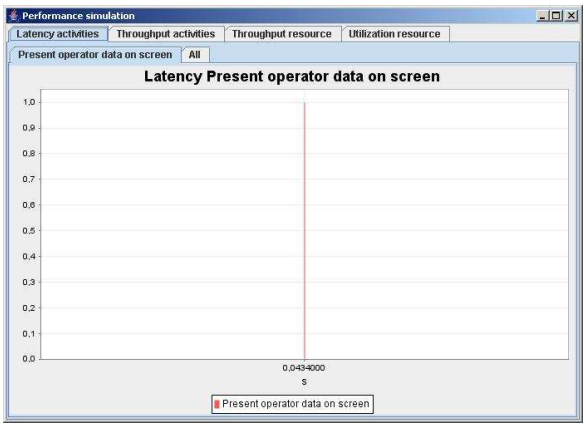

Figure 6.7: Example simulation run with only constant values

Figure 6.7 shows the total end-to-end latency of the *system* for the example. It (closely) corresponds to the calculations made on the analytical model: 20.28ms + 22.86ms = 43,14ms = 0,04314 seconds. The small difference between the example calculation and the result showed in figure 6.7 is due to rounding differences in the JAVA simulator, due to mixing of the integer, float and double datatypes.
The X-axis shows the absolute latency and the Y-axis shows the probability that this latency can occur. For only constant values the probability is (obviously) 1, or 100%.
Distributions can be set for each formula individually. Suppose that a normal distribution with a sigma(σ) of 1 is set for the *splice_network_src* formula, with a mean of 11.76ms. Running the simulation again leads to a different outcome, depicted in figure 6.8.

Figure 6.8 shows clearly a normal distribution. The average load remains at 0,0434 seconds, because this is the latency with the highest probability (0,0300). Furthermore, the X axis depicts 3 sigmas left from the mean, and on the right side two sigmas from the mean. The total latency is not a constant (probability of 1) anymore. The latencies shown in 6.8 is the latency that can occur some moment in time with a certain probability.
Interesting information can be extracted from figure 6.8. The mean lies approximately between 0,04300 and 0,0435. This means that all latencies between 0,04200s and 0,04400s will occur within 67% of all cases, because 1 sigma on the left and one sigma on the right spans 67%. A peak-load of greater than 0,4500 will occur in less than 5% of the time, as 0,4500 lies within more than 2 sigmas from the mean (and two sigmas span 95%).

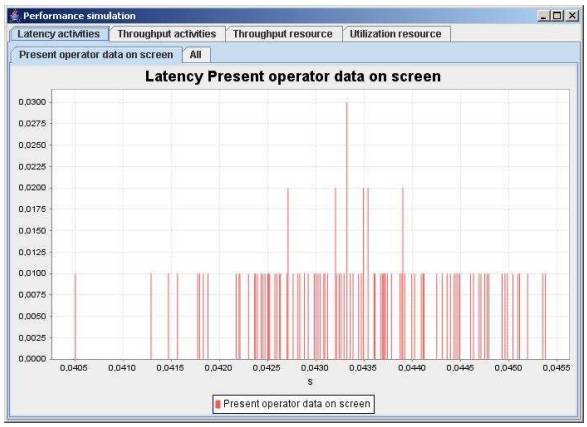

Figure 6.8: Example simulation run with a normal distribution

Distributions can also mix. When using more than one distribution, latency occurrences can occur in more than one distribution, resulting in higher probabilities. An example of this is depicted in figure 6.9, where an exponential distribution is set on the *splice_network_dst* formula, with an average of 21,84.

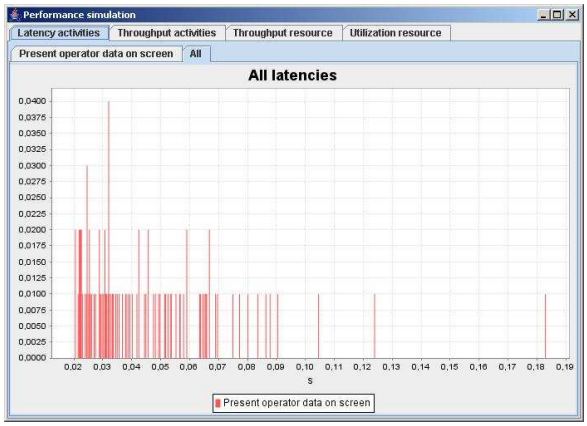

Figure 6.9: Example simulation run with a normal distribution and an exponential distribution

Figure 6.9 shows that the average latency still lies at approximately 0,35 seconds, but now with a probability of 0,0400. This is due to the fact that the normal distribution of figure 6.8 has now mixed with the exponential distribution. Another interesting observation of figure 6.9 is that the peak-latency that can occur is now more than 0,18 second: a giant increase. However, this will occur with a relatively small probability.

6.5 conclusion

The evaluation procedure presented in this chapter is twofold: it combines calculated results from UML system analysis models with simulation models by means of modeltransformations. Calculated latencies from behavioral flow diagrams are (automatically) incorporated in simulation models for characterizing the averages and sigmas of various distributions. These distributions are usually set by performance engineers. By running the generated, executable simulation model more details about the dynamics of the system can be obtained. These results are used in order to verify if the simulation outcome conforms to the expected calculated result: if it does, the simulation (and evaluation procedure) is considered successfull. Otherwise, system engineers need to change performance parameters or mappings and go back to the drawing board.

Part III

Implementation

Chapter 7

DESIDE

The proposed approach within this thesis for developing an integrated performance engineering method, is supported by a prototype tool, DESIDE. DEcision Support In a DEsign environment (DESIDE) is implemented in IBM® Rational Software Architect, and provides the system engineer modeling support and pre-configured transformations. The implementation follows the "one click on a button" principle, which means that feedback about the responsiveness of a system can be obtained directly by the provided automated tools. All actions are automated as much as possible.

7.1 Rational Software Architect

DESIDE is implemented in the IBM Rational Software product family [Ibm07]. This product family encompasses Rational Software Architect® , Rational Systems Developer®and Rational Software Modeler®. All products are based on Eclipse [Ecl07] and integrates UML Modeling and Modeltransformations with a JAVA-based development environment (IDE).
Currently, system engineers use the Rational product family in order to compose system models, using the provided UML modeling tools. For a developer, it is easy to add or extend the functionality of the Rational product family using the general Eclipse plugin structure: this plugin structure provides developers a MVC (model-view controller) design pattern [GHJV95], which separates the model from views. The existence of the Rational product family within Thales Netherlands B.V. and the possibility of adding extra functionality (easily) resulted in choosing IBM Rational Software Architect as the implementation technology of choice.

An import issue during the implementation in Rational Software Architect was the **separation of concerns(SoC)**. SoC means that a computer program is broken into distinct features that overlap functionality as little as possible. As a result, those functionalities can be assigned to roles. In this case, a role is someone that uses DESIDE and has a specific interest when using it. DESIDE currently supports the following roles:

1. *The system engineer* - composes the system and verifies performance results
2. *The performance engineer* - monitors running systems and analyzes operational data in order to build an accurate performance model for DESIDE

The separation of roles led to the development of two tools, both developed as IBM Rational plugins:

1. *a PerformanceAnnotationTool plugin* - that helps system engineers with composing and specifying systems
2. *a PerformanceTransformation plugin* - that evaluates system models and generates simulation models.

Both plugins are described in the next subsequent sections.

7.1.1 PerformanceAnnotationTool plugin

The PerformanceAnnotationTool plugin adds an performance view to the current Eclipse workbench of the system engineer. It provides easy access to stereotype attributes and provides editing possibilities without opening any extra windows (the standard RSA behavior). Figure 7.1 shows this view.

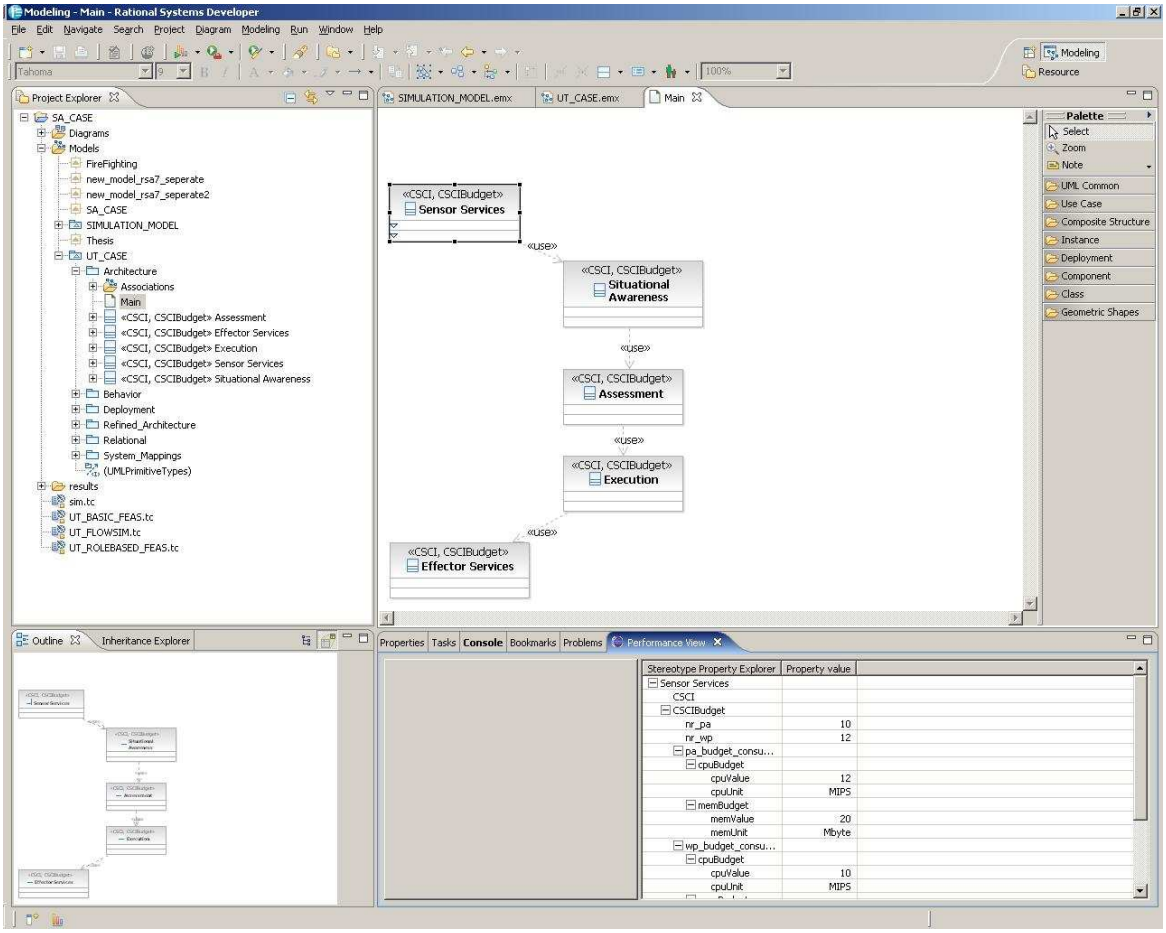

Figure 7.1: PerformanceAnnotationTool plugin screenshot

At the bottom of figure 7.1, the "Performance view" is shown. When the user clicks on an UML element that has an stereotype attached to it, the performance view immediately shows all composite properties. These properties can be edited directly. It is also possible to extend this plugin in order to predefine standard properties and values for the future. An overview of the JAVA structure of the plugin is depicted in figure 7.2.

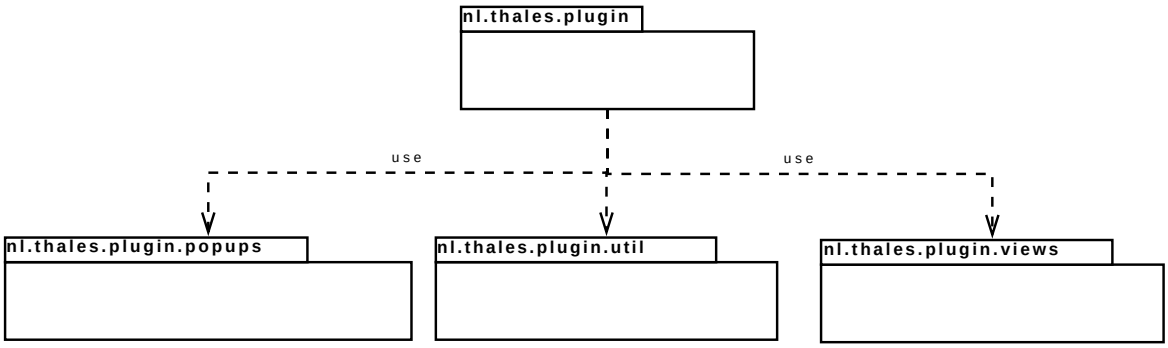

Figure 7.2: Overview of the PerformanceAnnotationTool structure

The basepackage of the PerformanceAnnotationTool plugin is the “nl.thales.plugin” package. This package contains the plugin manifest and the plugin entry class. The “nl.thales.plugin.popups” package contains extended menu-items, so any code that needs to be executed when the user clicks or right-clicks on an UML entity. The “nl.thales.plugin.util” package contains various helper classes such as EMF (Eclipse Modeling Framework), an UML implementation that provides UML elements e.g. classes, methods, stereotypes and stereotype attributes to the plugin developer. It prevents the programmer from directly modifying the XMI resources, which results in a maintainable solution. Finally, the “nl.thales.plugin.views” package contains the actual implemented views of the Eclipse model-view-controller pattern.

7.1.2 PerformanceTransformation plugin

The PerformanceTransformation plugin is an extensive plugin. It provides a set of modeltransformations to the system engineer that can be used to calculate and analyse UML models. It also presents the results directly to the user, by changing and adding visual elements (such as images) and UML elements (attributes that performance results) in the current models.
IBM Rational Software Architect uses the JAVA transformation API for their transformation engine. This approach distinguishes four important concepts:

1. *RootTransform* - The Root Transform class is an utility class that transformation authors can use as the root of their transformation. This transformation has a predefined structure: the initialization phase, the main phase and the finalization phase. The main phase can start other types of transforms, the initialization phase contains all code that have to be executed before the transformation starts, and the finalization phase contains all code that have to be executed after the transformation is finished.
2. *UMLKindTransform* - The UML kind Transform class is an utilization class especially designed for UML-based transformations. This transformation class provides easy access to UML elements.
3. *ModelRule* - A model rule is a utilization class that executes a transformation rule and generates a target model element out of a source model element.
4. *Condition* - A condition is an utilization class that filters the needed source elements out of a list of possible source elements that is available for processing. Typically, a condition is used in conjunction with a ModelRule to specify the source model element.

The four concepts mentioned above results in the following overview of a complete transformation:

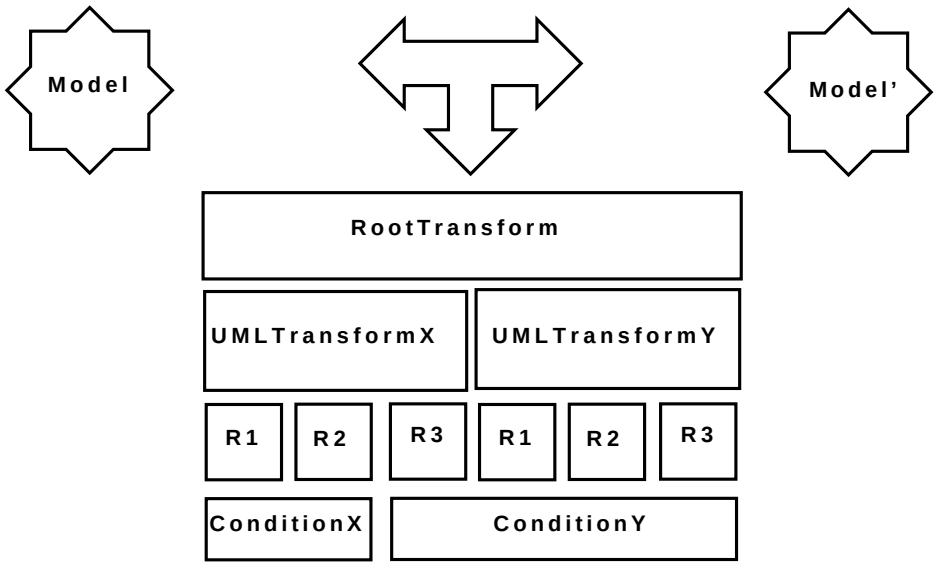

Figure 7.3: Overview of the transformation structure

Figure 7.3 shows that in order to transform from model to model prime, a RootTransform is executed that contains two transformations: UMLTransformX and UMLTransformY. Both transformations contain their own transformation rules, and each rule can contain one or more conditions that filters UML elements accepted for input.

An important issue when developing a transformation structure is the concept of *reuse*. Basically, reuse means using existing items more than once, which can be an UMLTransform, a rule or a condition. Two requirement for reuse are *high cohesion* and *low coupling*. A cohesive module means that all code is strongly related, whereas a low-coupled module means that the module is not dependent on other modules.
Most transformation work is performed in individual UML rules. A rule can be reused within separate transformations, but dependencies possibly exists. Input models or elements may not be the same, or a rule can depend on same kind of external data structure that can be received by other transformation rules via a *transformation context*. The transformation context is the shared environment where (JAVA) objects can be stored that can contain information valuable for transformation rules. However, the use of a global, shared environment does not stimulate the development of transformation objects that can be reused.
This thesis presents a structure that can be used when programming with transformations that manages dependencies among rules with the goal of stimulating reuse. This is achieved by introducing the *design by contract* principle, which concretely means that an extra interface is added to the ModelRule class with the goal of explicitly defining dependencies. This is depicted in figure 7.4.

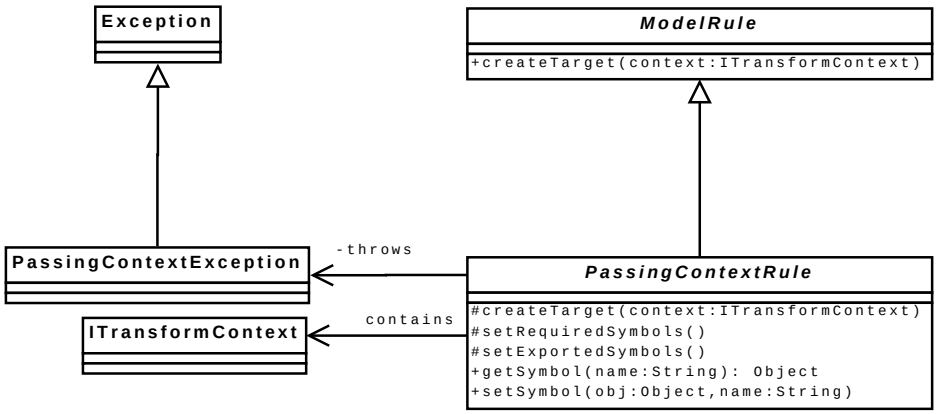

Figure 7.4: Designing by contract

Instead of directly using the *ModelRule* class, it is preferable to use the *PassingContextRule* class. This class extends the ModelRule class and its createTarget method. Every ModelRule class can directly use the ITransformContext environment, the environment where shared objects can be stored. The PassingContextRule ensures that if a ModelRule is defined that extends from PassingContextRule, instead of ModelRule, a list of *importedSymbols* and *requiredSymbols* is specified. Every time that a getSymbol or setSymbol method is called (retrieve or set objects from the shared environment), the symbol (name-based) is checked among both lists. If an object is used that is not explicitly specified via the required list, an exception is raised. Also, when an object is saved that is not explicitly specified via the export list, an exception is raised. Setting the required and exported symbols is a prerequisite for using this class, an exception is also raised if an programmer forgot about it. As a result, if a PassingContextRule is reused, it is always clear to everyone what is dependencies are.

Within this structure, and the current implementation, the following four transformations are defined:

1. *Basic feasibility analysis* - calculates if demanded resource budgets does not exceed supplied resource budgets. It uses basic hardware-software allocation schemes for calculating the final numbers
2. *Role-based feasibility analysis* - calculates if demanded resource budgets does not exceed supplied resource budgets. It uses role-workset-software allocation schemes for calculating the final numbers
3. *Behavioral flow feasibility analysis* - calculates if demanded responsiveness budgets does not exceed the demanded responsiveness budgets
4. *Simulation analysis* - Transforms system analysis models into simulation models

Using these four implemented transformations within RSA is fairly simple, as figure 7.5 (transformation configuration) shows.

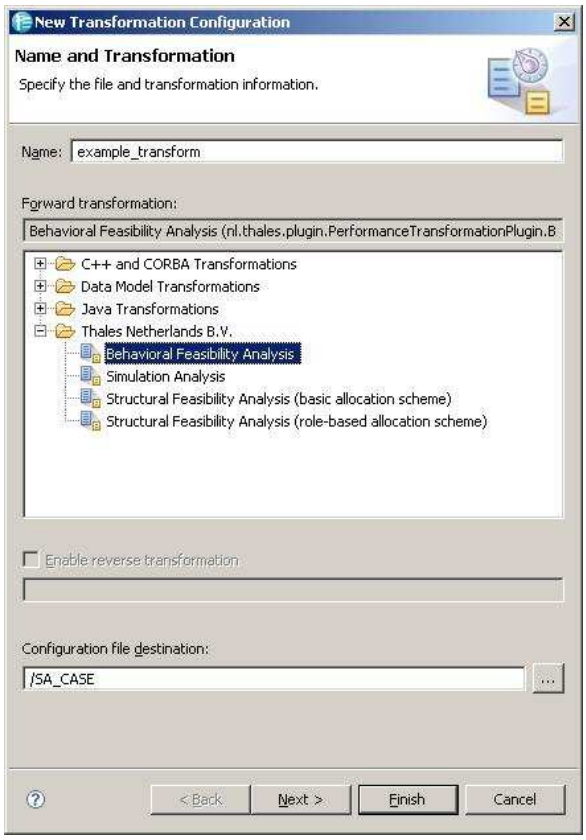

Figure 7.5: IBM Rational transformation configuration

System engineers can choose one of the four transformation depicted in figure 7.5. Furthermore, they have to specify a source model and a target model. When they are finished with composing the system model and annotating the system model with the necessary quality attributes, the run the transformation and obtain direct feedback. Feedback is provided in the model by means of generating extra visual elements and calculations results in the form of attributes. An example of this is depicted in figure 7.6 on the next page.

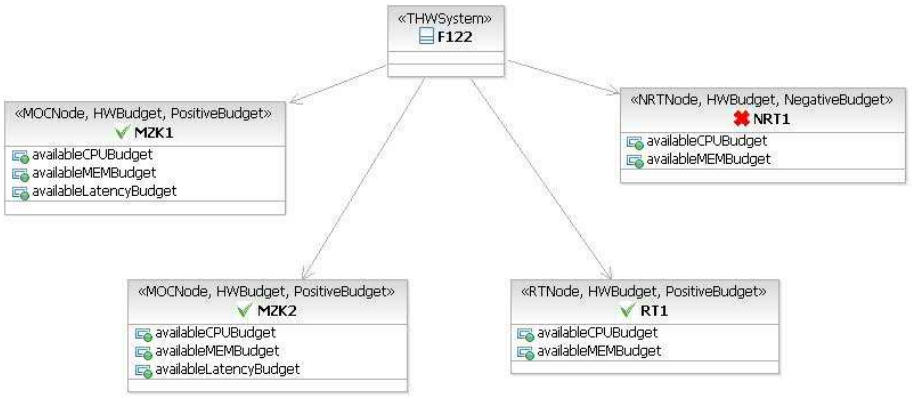

Figure 7.6: Example system after transformation run

Figure 7.6 shows an example of a deployment diagram after structural budget evaluation(resources) and behavioral budget evaluation(responsiveness budgets). In this example, three nodes(MZK1, MZK2, RT1) still provide sufficient resources (behavioral or structural), but one does not(NRT1), which is clearly depicted by the added red-cross in the model. The available budgets still left for allocation are presented as attributes in the target UML model, which can be inspected by system engineers. For a more detailed overview, system engineers can consult the generated textfile.

The last important issue when building reusable, flexible transformations comprises the implementation of the calculation engine. Chapter 6 already described the evaluation rules that were executed when assessing responsiveness. These rules were based on formulas derived from real operation data [Tha07]. However, these rules are subject to change, because Thales Netherlands B.V. continuously measures its system under various environments. New insights and measurements leads new formulas. These formulas (which are used by the transformation plugin) are maintained by performance engineers, and change frequently.
During this research thesis, a dedicated ANTLR parser [Ant07] was developed in order to provide a flexible method that can specify calculations in a non-ambiguis way. This parser can interpret "calculation files", where calculations are specified using local variables or profile variables. Profile variables represent metamodels attributes and provide a way to connect calculations with elements of the developed metamodel. An example of such a calculation file is listed below:

```
//IMPORTED VARS
localvar import_platform:                                    float;
        //0 = AMD-OPTERON, 1=POWER-PC
localvar import_nr_consumers:                                float;

//RULES
localvar rule_amd_basicwrite_local:                          float;
localvar rule_ppc_basicwrite_local:                          float;
localvar rule_basicwrite_local:                              float;
localvar result:                                             float;

//       PROFILEVARS
profilevar DatastoreBudget.attributes.data_entity_size:      float;
profilevar LoadConstraint.loadFrequency:                     float;

//BEGIN PROGRAM
rule_amd_basicwrite_local := 6 + (7 + 4 * import_nr_consumers)
                + (10 * DatastoreBudget.attributes.data_entity_size);
rule_ppc_basicwrite_local := 18 + (21 + 12 * import_nr_consumers)
                + (30 * DatastoreBudget.attributes.data_entity_size);
rule_basicwrite_local := 0;
```

```
rule_basicwrite_local := if (import_platform == 0)
                then (rule_basicwrite_local + rule_amd_basicwrite_local)
                else (rule_basicwrite_local + rule_ppc_basicwrite_local);
//result is in ms...
result := (rule_basicwrite_local
                * LoadConstraint.loadFrequency) / 1000;
```

An overview of the EBNF grammar is presented in appendix A.

The final structure of the "PerformanceTransformationPlugin" is depicted in figure 7.7 on the next page. This figure depicts that the mainpackage, "nl.thales.plugin", consists of three subpackages: "nl.thales.plugin.calc", "nl.thales.plugin.transform" and "nl.thales.plugin.util". The "plugin" package contains the manifest and initial startup classes. The "calc" package contains all information related to the calculation process: it contains various structures for calculating UML model elements and it contains the dedicated ANTLR parser. The "transform" package contains all UML rules, exceptions, conditions and transformations that are necessary to implement the four specified transformations. The final package, the "util" package, contains a "nl.thales.plugin.util.resources" packages, which contains resources such as external libraries, images, calculation files with formulas and global configuration files.

7.2 Conclusion

In this chapter the prototype tool was presented. It is implemented as a set of IBM Rational plugins, which provides a "clean" plugin structure, based on the model-view-controller pattern provided by Eclipse.
The IBM Rational product family uses the JAVA transformation API as their transformation engine. This has a number of drawbacks as opposed to e.g. QVT: all transformation code has to be specified in an imperative language (JAVA), whereas QVT provides means to select model entities by one functional statement. This prevents a lot of repetitive code. Another drawback of the JAVA transformation API is that managing dependencies among transformation rules is hard. However, a structure was presented that stimulates reuse by incorporating the design-by-contract principle.
Finally, in order to specify and manage fomulas that are used for calculation purposes, a dedicated ANTLR parser was developed that could manage and interpret calculation files in a non-ambigous way. This construction connects calculations with the previous developed metamodel, by means of introducing profile-variables which represents meta-model attributes. When a system engineer specifies such a variable, the implementation automatically walks through all elements of the source model and collects all specified elements. The value of the element is automatically inserted for the relevant calculation. All presented implementation constructions provide a high degree of automation to the system engineer.

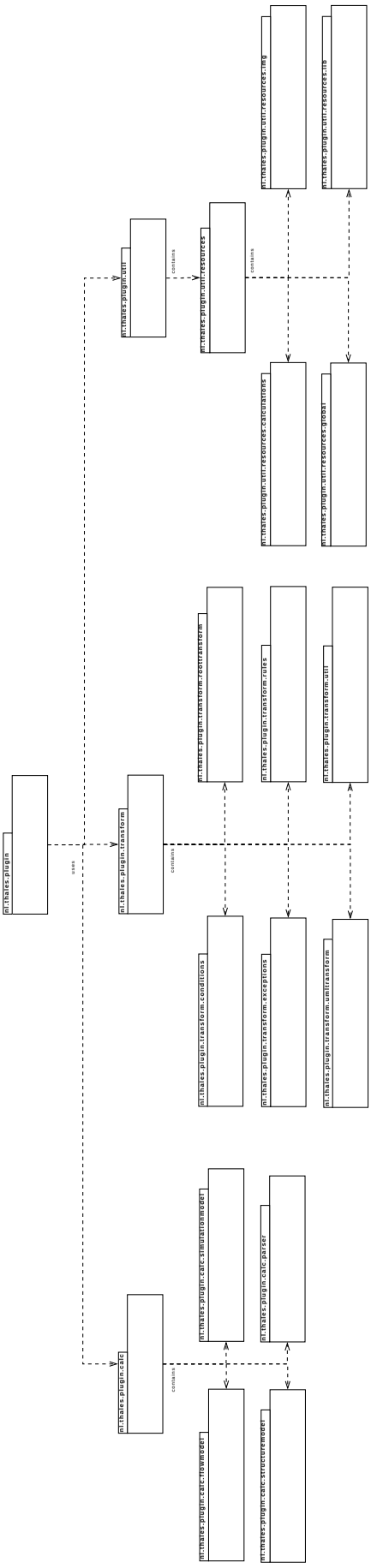

Figure 7.7: Overview of the PerformanceTransformationPlugin structure

Part IV

Evaluation

Chapter 8

Conclusion

This chapter concludes the thesis. It reflects back on achieved results, presents additional discussion and discusses related work.

The main goal of this research thesis was to develop an integrated method for analyzing and predicting responsiveness of the Combat Management System for Thales Netherlands B.V. It started in chapter 2 with the investigation of the current system development process and the identification of (possible) bottlenecks. This resulted in two important observations: performance information was scattered among system engineers and the specification of available resources (structural and behavioral) is a prerequisite because of the limited space in a naval vessel. These two observations formed the motivation for developing an integrated performance engineering method for early analysis and prediction of responsiveness.
Next, a literature study was conducted that searched for additional background information about performance engineering in general and investigated a number of responsiveness definitions. Also, proven evaluation techniques were considered, but unfortunately none of them was directly usable, because the gap between a system model and a (formal) analysis model is too big. This resulted in a model-based approach.
The final solution presented within this thesis uses model-artefacts produced during the regular system development process. As a result, a high level of integration could be achieved. Within the proposed solution, meta-models were presented that described exactly the structure of systems and performance models. Furthermore, transformations were presented that could generate intermediate models, with the goal of obtaining an executable simulation model. This closed the gap between UML system models and formal system analysis models. Analytical evaluation procedures were used for calculating the responsiveness of the system and simulation based methods were used for verifying the outcome of the analytical models.
Finally, this thesis presented a prototype tool that provides a high degree of automation to system engineers. This was a key success factor for successfully applying performance engineering within industry.

8.1 Answers to research questions

The main research question was stated as:

What process- and product changes are necessary in order to be able to perform early analysis and prediction of responsiveness for the Combat Management System (CMS) used within Thales Netherlands B.V. ?

With regards to the **process** used within Thales Netherlands B.V. little changes are necessary to the current process. Thales Netherlands B.V. uses an implementation of the system engineering process, which generally means that performance assessment is performed after the integration phase. If early analysis and prediction of responsiveness is a goal, then responsiveness assessment should be performed after the modeling phase, early during the overall process. When the

evaluation-procedure of the validation step concluded that the responsiveness of the composed system is not satisfiable, models can be changed accordingly. This prevents expensive changes afterwards.
The **product** based changes were extensive. This thesis presented a model-based approach with the goal of obtaining an executable simulation model is able to calculate and to verify the responsiveness of the composed system. However, model-based approaches directly imply the availability of metamodels. Unfortunately, existing system metamodels or performance metamodels were not available, so both models needed to be developed. By combining the models, and the formulas obtained from operational measurements, implemented by a dedicated ANTLR parser a highly automated tool could be developed that helps system engineers with measuring and verifying responsiveness of system models. Verification is achieved by transforming a simulation model out of an UML system analysis model and feeding this model into a discrete-event JAVA simulator.

The first subquestion was:

What is responsiveness?

Responsiveness is a very "broad" definition and the notion of responsiveness differs among current literature. However, some consensus do exists, as in every definition the notion of "timeliness" seems to appear. The most clear, and workable definition of responsiveness is 'the ability of a system to meets its objectives for response time or throughput' [Smi02].
However, determining the responsiveness of a system is not a trivial task. Operational laws, such as Little's law can be used for determining the responsiveness of an abstract system, but it is hard to apply this on large, complex systems where many factors have to be taken into account. Building such a closed-form analytical (theoretical) solution is a time-consuming, almost impossible task.
Instead, a relatively simple analytical solution is built for systems of Thales Netherlands B.V. that makes use of real operational data. By combining concepts of model-driven engineering, such as automation, abstraction and transformation engines, along with formulas that describe the responsiveness of the middleware usage, a performance model could be obtained. Therefore, in this thesis, responsiveness is defined as the total *end-to-end latency of a functional flow that can occur within a system.*

The second subquestion was:

Are there general methods and frameworks available within current literature that address performance and responsiveness?

There are methods and frameworks available within current literature that address performance related issues. This thesis looked into software performance engineering, a systematic and quantative approach that supports early performance analysis and prediction. It is, however, only a process-based framework: it only describes the required activities for performance analysis and prediction. It does not provide details about the introduced software execution model and the system execution model, it only describes the general concepts and what activities should be performed during the performance engineering process. Reasoning frameworks provide more detail about models and their relations, but they do not provide any semantics. However, recent developments within the OMG led to the development of MARTE, a successor of the UML SPT Profile, where detailed semantics are provided. Because of the lack of one general framework that is able to express all the concepts mentioned above, this thesis combined concepts of all methods.

The third subquestion was:

What is the impact of responsiveness on system design?

System design has a serious impact on responsiveness. A degraded system performance due to responsiveness problems can lead to changes in system design. It is therefore important that system engineers have enough possibilities to change things without having to change the internals of the system, because this usually introduces unnecessary risks. This thesis introduced the notion of mappings, which provides an easy way to e.g. change the allocation structure of hardware and

software. Mappings are also used for assigning behavioral diagrams to architectural components and within these diagrams, the exact responsiveness numbers can be calculated. By changing various mappings, system engineers can influence the responsiveness of the composed system heavily.

The fourth subquestion was:

How can consistency be achieved between system design and performance models?

Generally, system concepts and system analysis concepts do not map 1-on-1 to each other. This is called the "gap" between system models and formal system analysis models, already described in one of the answers above. By using intermediate models during the transformation process with well-defined transformation rules, consistency can be achieved among those models.

The fifth subquestion was:

How to built a consistent set of tools that supports the models used within a typical system engineering process?

A consistent set of tools has to support the overall software performance process: from composing systems to annotating models and the ability to evaluate the composed system, that provides direct feedback to system engineers, all *from the same environment*. The IBM Rational Software Architect environment provides a clean, Eclipse based plugin structure based on a model-view-controller design pattern, that can be used to extend or add functionality. The prototype developed within this thesis supports all mentioned activities by means of a set of plugins, that can all be used and executed from the same environment.

The sixth subquestion was:

How can obtained performance results be verified?

An important issue within the overall performance engineering process is verifying if obtained results are correct. Within the proposed solution, simulation is used as a verification step, because a simulation can provide insight in aspects of the dynamic behavior of the system that cannot be obtained when using closed form, analytical solutions. The most important information of the presented simulation is the information that can be extracted from distributions. Distributions can be used to describe the occurrence of peak-latencies over time, with a certain probability. This is information that can be used to verify the certainty of the outcome of the analytical evaluation procedure (calculation). By combing analytical results with simulation, the trustworthiness of the results can be assessed.

8.2 Related work

There is a substantial amount of related work available for this subject. Examples include the investigation of a common software ontology [Cor05], the description of several design features that can impact the performance of distributed applications [RVH95], and tranforming analytical models, such as a queuing network, from UML specifications [PS02].

An interesting framework, discussed in section 3.3.2, is the reasoning framework from the Software Engineering Institute (SEI). This framework provides a "black-box approach" to non-experts. It is a rather "static" framework, but if a reasoning framework was incorporated for describing the overall solution, it would look like figure 8.1 depicted on the next page.

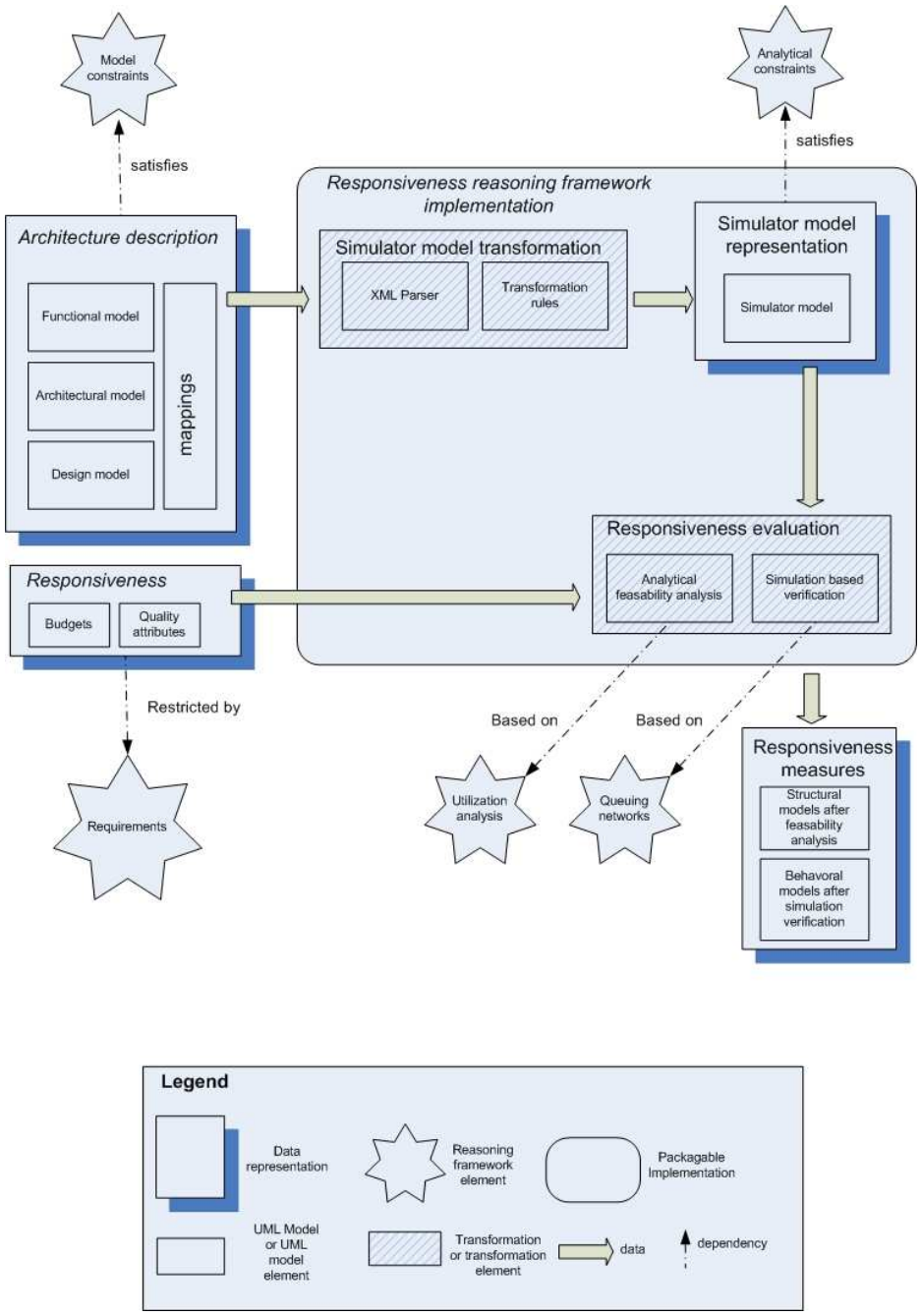

Figure 8.1: A model-based performance prediction reasoning framework

Figure 8.1 emphasizes the following:

1. It *extends* the architectural description with models
2. It *calculates* the resource utilization among annotated system models
3. It *verifies* the occurrence of peak-latencies
4. It provides *feedback* to the system engineer

The reason why a reasoning framework is not incorporated for describing the overall solution is that it does not shows issues such as "the gap" between system models and formal analysis models, which is something that this thesis is explicitly solving.

8.3 Recommendations and future work

This research thesis is a valuable contribution for Thales Netherlands B.V. in a number of ways. First, it really presents an integrated performance engineering method, with a consistent set of tools. Second, it contributes towards specifying the structure of systems by means of a number of developed meta-models, which where non-existing. Third, this method closes the gap between system models and formal system analysis models, a concept acknowledged within current literate but difficult to solve. Solving this gap concretely means that system engineers can use their regular UML models for system composition, without losing (formal) verification capabilities.

However, there exists a number of limitations within this research. Interpreting the outcome of simulation runs is a far from trivial process. Calculations made with multiple distributions can lead to mixed distributions, which basically leads to results that can be unexplainable. The value of simulation results then has to be estimated by experts. The research of automated feedback, on calculated models or simulation models, can be subject of further research.

Furthermore, a recommendation is to investigate and define a number of critical quality concerns, that prevents the system engineer from manually annotating system models. A lot of performance parameters have to known in advance in order to obtain a realistic understanding of the final system behavior, and it is not always realistic to assume that the system engineer knows all the numbers in advance. Therefore, if quality concerns are defined that can be used across various abstraction levels, system engineers only have to characterize a system instead of annotating all the numbers by hand. The composite pattern used within the developed metamodels already supports the grouping of quality attributes, but more research should be put into defining and identifying the most critical quality concerns.

Another recommendation is the usage of this method and the developed tool in a real project in order to extensively test the usability of the tool and the method. Valuable feedback can be obtained that could improve the quality of the overall work.

Finally, there are a number of recommendations regarding the implementation specifically. A drawback of using Rational Software Architect is that it does not provide good modeling support for OCL constraints. It only supports the parsing of one OCL statement attached to a model element, and checks only if a constraint is syntactically valid. The constraints of the developed metamodel requires the execution of calculations, because they could be imposed on the result of the calculations. However, this is a requirement that is currently unachievable when using Rational Software Architect. The functionality of those constraints are currently implemented by a set of stereotypes, with some additional JAVA code that is executed after the calculation-transformation is finished. If OCL support improves, a recommendation is to use OCL for constraint specification instead of stereotypes with JAVA code.
Also, the current incorporated simulator has its limitations: it is e.g. not possible to define workloads (frequency of produced / consumed items) within subdiagrams (callbehavior actions) of activity diagrams. This means that either an activity diagram and all subdiagrams have a fixed workload, or the simulation strucure is flattened which leads to scattered simulation results. This is a major drawback of the current simulator, so the incorporation of alternate simulators needs to be researched.

Appendix A

EBNF grammar

A.1 CalcParser grammar

```
// CHANGES
//       2003.04.04  ruys                 Started (VB 2003).
//       2004.04.12  ruys                 Modifications for VB 2004.
//       2006.04.22  ruys                 Updated for Java 5.
//       2007.09.17  s0117587  Modified for Performance Calculations
header {
    package nl.thales.plugin.calc.parser;
    import java.io.*;
}

// {{{--- Parser ------------------------------------------------------------

class CalcParser extends Parser ;

options {
    k = 1;                                  // # tokens lookahead
    exportVocab = Calc;                     // call the vocabulary "Calc"
    buildAST = true;                        // build an AST
    defaultErrorHandler = false;
}

// Imaginary tokens that are used for AST construction.

tokens {
    PROGRAM_AST;
}

// {{{--- Parser: production rules ------------------------------------------

program
  : code EOF!
    { #program = #([PROGRAM_AST, "program"], #program) ; }
    // Without a special root node, the ASTFrame will only
    // contain the first explicit node.
  ;

code
  : ((declaration)* (statement))+
  ;
```

```
declaration
  : ((LOCALVAR^ | PROFILEVAR^) IDENTIFIER COLON! type SEMICOLON!)
  ;

statement
  : expr SEMICOLON!
  | printStatement SEMICOLON!
  ;

expr
  : exprI (BECOMES^ expr)?
  ;

exprI
  : exprN
  | IF^ LPAREN! exprN RPAREN! THEN! LPAREN! expr RPAREN! ELSE! LPAREN!
    expr RPAREN!
  ;

exprN
  : exprLowPrecedence ((LESS^ | LESSEQ^ | GREATER^ | GREATEREQ^
  | EQUAL^ | NEQUAL^) exprN)?
  ;

exprLowPrecedence
  : exprHighPrecedence ((PLUS^ | MINUS^) exprLowPrecedence)?
  ;

exprHighPrecedence
  : operand ((MULTIPLY^ | DIVIDE^)operand)*
  ;

printStatement
  : PRINT^ LPAREN! expr RPAREN!
  ;

operand
  : IDENTIFIER
  | NUMBER
  | LPAREN! expr RPAREN!
  ;

type
  : FLOAT
  | INTEGER
  ;
// }}}--- Parser: production rules
// }}}--- Parser
```

A.2 CalcLexer grammar

```
// {{{--- Lexer ----------------------------------------------------------

class CalcLexer extends Lexer;

options {
    k = 2;                              // needed for WS (for instance)
    charVocabulary = '\3'..'\377';      // use ASCII character set
    exportVocab = Calc;                 // call the vocabulary "Calc"
    caseSensitive = true;               // lower and upper case is significant
    caseSensitiveLiterals = true;       // literals are case sensitive
    testLiterals = false;               // do not check the tokens table
}

tokens {
    LOCALVAR    =       "localvar" ;
    PROFILEVAR  =       "profilevar";
    PRINT       =   "print" ;
    INTEGER     =   "integer" ;
    FLOAT       =       "float";
    IF          =       "if";
    THEN        =       "then";
    ELSE        =       "else";
}

// operators

BECOMES     :   ":=" ;
PLUS        :   '+' ;
MINUS       :   '-' ;
DIVIDE      :   '/' ;
MULTIPLY    :   '*';
COLON       :   ':' ;
SEMICOLON   :   ';' ;
LPAREN      :   '(' ;
RPAREN      :   ')' ;
LESS        :   '<' ;
LESSEQ      :   "<=" ;
GREATER     :   '>' ;
GREATEREQ   :   ">=" ;
EQUAL       :   "==" ;
NEQUAL      :   "!=" ;

protected
LOWER       :   ('a'..'z') ;

protected
UPPER       :   ('A'..'Z') ;

protected
DIGIT       :   ('0'..'9') ;

protected
DOT         :   '.' ;

protected
DOLLAR      :   '$';
```

```
// The option testLiterals is only set to true for the rule IDENTIFIER!
// Consequently, after matching the rule, ANTRL will look in the tokens
// table to check whether the string found is an IDENTIFIER or a TOKEN.
IDENTIFIER
  options { testLiterals = true; }
  : (DOLLAR | LOWER | UPPER | DOT | '_' )
    (DOLLAR | LOWER | UPPER | DOT  | '_' | DIGIT )*
  ;

NUMBER
  : (DIGIT)+ (DOT (DIGIT)+)?
  ;


// Comments -- ignored
COMMENT
  :  "//" (~'\n')* '\n'
  { newline(); $setType(Token.SKIP); }
  ;

//Whitespace -- ignored
WS
  : ( ' '
  |   '\t'
  |   '\f'
  // handle newlines
  |  (  "\r\n"  // Evil DOS
     |  '\r'    // Macintosh
     |  '\n'    // Unix (the right way)
     )
  { newline(); }
  )
  { $setType(Token.SKIP); }
  ;

// }}}--- Lexer
```

A.3 CalcChecker grammar

```
// {{{--- CalcChecker ----------------------------------------------------

{ import java.util.*; }

class CalcChecker extends TreeParser;

options {
    buildAST           = true;  // AST does not have to be transformed
    defaultErrorHandler = false; // turn-off ANTLR's error handler to
                                 // propagate CalcExceptions to main
}

{
  ArrayList idset       = new ArrayList();
  ArrayList profilevars = new ArrayList();

  public boolean isDeclared(ArrayList list, String s)
  {
    return list.contains(s);
  }
  public void declare(ArrayList list, String s)
  {
    list.add(s);
  }
}

program
  : #(PROGRAM_AST (code)?)
  ;

code
  : ((declaration)* (statement))+
  ;

declaration
  : (localdeclaration | profiledeclaration)
  ;

localdeclaration
  : #(LOCALVAR id:IDENTIFIER type)
  {   if (isDeclared(this.idset, id.getText()))
        throw new CalcException(id.getText() + " is already declared");
      else
        declare(this.idset, id.getText());
  }
  ;

profiledeclaration
  :  #(PROFILEVAR id:IDENTIFIER type)
  {    if (isDeclared(this.profilevars, id.getText()))
         throw new CalcException(id.getText() + " is already declared");
       else
         declare(this.profilevars, id.getText());
  }
  ;
```

```
statement
  :     expr
  |     #(PRINT expr)
  ;

expr
  :   exprI
  |     #(BECOMES id:IDENTIFIER expr)
  {   if (!isDeclared(this.idset, id.getText()))
        throw new CalcException(id.getText() + "is_not_declared");
  }
  ;

exprI
  : exprN
  | #(IF exprN expr expr)
  ;

exprN
  : exprLowPrecedence
  | #(LESS exprLowPrecedence exprLowPrecedence)
  | #(LESSEQ exprLowPrecedence exprLowPrecedence)
  | #(GREATER exprLowPrecedence exprLowPrecedence)
  | #(GREATEREQ exprLowPrecedence exprLowPrecedence)
  | #(EQUAL exprLowPrecedence exprLowPrecedence)
  | #(NEQUAL exprLowPrecedence exprLowPrecedence)
  ;

exprLowPrecedence
  : exprHighPrecedence
  | #(PLUS exprLowPrecedence exprLowPrecedence)
  | #(MINUS exprLowPrecedence exprLowPrecedence)
  ;

exprHighPrecedence
  : operand
  | #(DIVIDE exprLowPrecedence exprLowPrecedence)
  | #(MULTIPLY exprLowPrecedence exprLowPrecedence)
  ;

operand
  : id:IDENTIFIER
  {  if ( (!isDeclared(this.idset, id.getText()))
       && (!isDeclared(this.profilevars, id.getText())) )
         throw new CalcException(id.getText() + "is_not_declared");
  }
  |   n:NUMBER
  ;

type
  : FLOAT
  | INTEGER
  ;

// }}}--- CalcChecker
```

A.4 CalcInterpreter grammar

```
header {
    package nl.thales.plugin.calc.parser;
    import java.io.*;
    import java.util.*;
}

class CalcInterpreter extends TreeParser;

options { importVocab = Calc; }

{
  private HashMap<String, String> store       = new HashMap<String,String>();
  private HashMap<String, String> profileVars = new HashMap<String,String>();

  public HashMap<String, String> getCalculationMap() {
    return this.store;
  }
  public void setProfileValue(String key, String value) {
    profileVars.put(key, value);
  }
}

program
  : #(PROGRAM_AST (declaration | statement)+)
  ;

declaration
  :     (localdeclaration | profiledeclaration)
  ;

statement { float v = (float)0; }
  :     expr
  | #(PRINT v=expr)
  { System.out.println("value_is:" + v); }
  ;

profiledeclaration
  : #(PROFILEVAR id:IDENTIFIER type)
  {
    String value = profileVars.get(id.getText());
    if ( value != null) {
      store.put(new String(id.getText()), value);
    }
    else {
      store.put(new String(id.getText()), Float.toString(new Float(-1)));
    }
  }
  ;

localdeclaration
  : #(LOCALVAR id:IDENTIFIER type)
  { store.put(new String(id.getText()), Float.toString(new Float(-1))); }
  ;
```

```
expr returns [float val] { float v; val = (float)0; }
  : val = exprI
  | #(BECOMES id:IDENTIFIER v=expr)
  {
    store.put(new String(id.getText()), Float.toString(new Float(v)));
    val = v;
  }
  ;

exprI returns [float val] { float x, y, z; val = (float)0; }
  : val = exprN
  | #(IF x=exprI y=expr z=expr)
  {
    if (x != 0) { val = y; }
    else { val = z; }
  }
  ;

exprN returns [float val] {float x, y, z; val=(float)0;}
  : val = exprLowPrecedence
  | #(LESS x=expr y=expr)
    { System.out.println("less"); if(x<y) val=1; else val = 0; }
  | #(LESSEQ x=expr y=expr)
    { System.out.println("lesseq");if(x<=y) val=1; else val = 0; }
  | #(GREATER x=expr y=expr)
    { System.out.println("greater");if(x>y) val=1; else val = 0; }
  | #(GREATEREQ x=expr y=expr)
    { System.out.println("greatereq");if(x>=y) val=1; else val = 0; }
  | #(EQUAL x=expr y=expr)
    { System.out.println("equal");if(x==y) val=1; else val = 0; }
  | #(NEQUAL x=expr y=expr)
    { System.out.println("nequal");if(x!=y) val=1; else val = 0; }
  ;

exprLowPrecedence returns [float val] { float x, y; val = (float)0; }
  : val=exprHighPrecedence
  | #(PLUS x=exprLowPrecedence y=exprLowPrecedence)   { val = x + y; }
  | #(MINUS x=exprLowPrecedence y=exprLowPrecedence)  { val = x - y; }
  ;

exprHighPrecedence returns [float val] { float x, y; val = (float)0; }
  : val=operand
  | #(DIVIDE x=exprLowPrecedence y=exprLowPrecedence)   { val = x / y; }
  | #(MULTIPLY x=exprLowPrecedence y=exprLowPrecedence) { val = x * y; }
  ;

operand returns [float val = (float)0]
  : id:IDENTIFIER { val = Float.parseFloat(store.get(id.getText())); }
  | n:NUMBER      { val = Float.parseFloat(n.getText()); }
  ;

type
  : FLOAT
  ;

// }}}--- CalcInterpreter
```

Bibliography

[Ant07] http://www.antlr.org, 2007. ANTLR website.

[BBS02] S. Balsamo, M. Bernardo, and M. Simeoni. Combining stochastic process algebras and queuing networks for software architecture analysis. In *ACM Proceedings of the International Workshop Software and Performance, 109-202*, 2002.

[BDMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. In *IEEE Transactions on Software Engineering, Vol30, No5*, 2004.

[BIKM05] L. Bass, J. Ivers, M. Klein, and P. Merson. Reasoning frameworks. Technical report, Software Engineering Institute, 2005.

[BJR99] G. Booch, I. Jacobsen, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Buz76] J.P. Buzen. *Fundamental Operational Laws of Computer System Performance*. Acta Informatica, 1976.

[Cor05] Vittorio Cortelessa. How far are we from the definition of a common software performance ontology. In *ACM Proceedings of the International Workshop Software and Performance*, 2005.

[Des07] http://www.desmoj.de, 2007. DesmoJ website.

[Ecl07] http://www.eclipse.org, 2007. Eclipse website.

[ESS07] http://www.cse.dcu.ie/essiscope/sm2/9126ref.html, 2007. ESSI-SCOPE website.

[Gee07] Boudewijn Geerink. System architecture description of the tacticos combat management system based on dodaf. Master's thesis, University of Utrecht, 2007.

[GHJV95] Erich. Gamma, Richard. Helm, Ralph. Johnson, and John. Vlissides. *Design Patters, Elements of Reusable Object-Oriented Software*. Addison-Wesly, 1995.

[Gro95] The Standish Group. The chaos report, 1995.

[Gro05] The Object Management Group. Uml profile for schedulability, performance and time. Technical Report 1.1, The Object Management Group, 2005.

[Gro07] The Object Management Group. A uml profile for marte. Technical Report Beta 1, The Object Management Group, 2007.

[GS07] Maurice Glandrup and Rene Scholte. Budget-driven performance modeling in software intensive systems, 2007.

[Hoo06] Niek Hoogma. Modeling performance indicators using the spt and sysml profiles. Master's thesis, University of Twente, 2006.

[Hor06] Ivo Horst, Ter. Deside, decision-support using system simulation in a design environment, 2006.

[Hor07] Ivo Horst, Ter. Performance evaluation in an early development phase. Master's thesis, University of Twente, 2007.

[Ibm07] http://www.ibm.com/software/rational, 2007. IBM Rational Software product family website.

[IEE07] http://www.ieee.org, 2007. IEEE website.

[Inc07] http://www.incose.org/practice/whatissystemseng.aspx, 2007. Internation Council on System Engineering website.

[Iso07] http://www.iso.org, 2007. Internation Standardization Group website.

[Kru95] P. Kruchten. Architectural blueprints - the 4+1 view model of software architecture. *dde*, 1995.

[Lit61] J.D.C. Little. A proof of the queuing formula l = λ w. In *Operations Research 9, 383-387*, 1961.

[omg07] http://www.omg.org, 2007. Object Management Group website.

[Pag84] Ernest H. Page. *Simulation Modeling Methodolgy: Principles and Etiology of Decision Support*. PhD thesis, Virginia Polytechnic Institute and State University, 1984.

[Pri07] http://www.prismtech.com, 2007. PrismTech website.

[PS02] C. Dorina Petriu and Hui Shen. Applying the uml performance profile: Graph grammar-based derivation of lqn models from uml specifications, 2002.

[RVH95] J. Rolia, V. Vetland, and G. Hills. Ensuring responsiveness and scalability for distributed applications. In *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, 1995.

[Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE magazine*, 2006.

[Sel04] B Selic. Summer school mdd for dres, 2004. Presentation slides.

[Smi02] Connie U. Smith. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

[SW01] Ansgar Schleicher and Bernhard Westfechtel. Beyond stereotyping: Metamodeling approaches for the uml. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.

[Tha07] Thales Netherlands B.V. *Guidelines for application editors, performance manual*, 2007.