Master's Thesis

in Computer Science

# Performance Evaluation in an Early Development Phase

I.J. ter Horst

February 2007

Committee

| | |
|---|---|
| Dr. D.N. Jansen | University of Twente |
| Drs. A.G. Kleppe | University of Twente |
| Dr. M.I.A. Stoelinga | University of Twente |
| Ir. M.H.J. Glandrup | Thales Nederland |
| Dr. Ir. R.C. Scholte | Thales Nederland |

THALES

University of Twente
Enschede - The Netherlands

# Performance Evaluation in an Early Development Phase

I.J. ter Horst

February 2007

— unclassified —

Thales Nederland B.V.
Haaksbergerstraat 49
7554 PA Hengelo
The Netherlands

University of Twente
Drienerlolaan 5
7522 NB Enschede
The Netherlands

This thesis was typeset using $\LaTeX\,2_\varepsilon$.

# Abstract

In the area of performance evaluation, a lot of research has been done on real-time constraints such as communication latencies and throughput requirements. However, almost no research has been conducted on performance evaluation in a very early development phase. The study presented in this thesis provides a method that can be applied to do performance evaluation in an early development phase, even when the structure of software and hardware are just being defined.

Performance evaluation of combat management systems at Thales Naval Nederland is currently performed too late in the development process. Failure to meet performance goals can be costly and should be avoided, so early discovery of problems is important. Instead of the calculations in large Excel sheets which were used, this thesis presents a UML model with which the structure and performance information of a system can clearly be defined.

To support an engineer in defining the structure of software and hardware, the well-known modeling language UML is used. To specify the systems developed by Thales, we have created number of stereotypes and grouped them in the new UML profile *TProfile*. The stereotypes define default elements an engineer can use to define the structure of software and hardware in a hierarchical way. Allocation relationships link software to hardware and define which parts of the software run on which hardware parts.

Performance information can be added to all elements of the structural model of a system in terms of budgets. A budget defines an amount of resources provided by the hardware or required by the software system. We have defined a small expression-based language to express budget values, which are either fixed, or can depend on other budgets. We thus provide the possibility to express dependencies between different parts of the system.

Constraints are defined on allocation relationships, where the software meets the hardware. A constraint relates the budget of a software element to the budget of a hardware element and defines a restriction on that relationship. This allows engineers to express whether the software system fits on the hardware system.

As the UML models of systems tend to grow large and a lot of budgets and constraints are added, evaluation of the constraints becomes a hard task. Automatic constraint verification is therefore provided by our System Verifier Tool, which integrates with Rational Software Architect (RSA), the tool used within Thales to create UML models. We use RSA's UML modeling functionality of RSA and introduce the TProfile with our System Verifier Tool, which recognizes the elements relevant to Thales by means of the stereotypes. Because of the integration of the System Verifier Tool in RSA, verification of all constraints created for a system can be done at the push of a button. Verification results are presented clearly to the engineer.

The System Verifier Tool allows engineers to easily experiment with the structure of hardware and software, budget values and constraints and thus supports system development in an early phase.

“Computer Science is no more about computers
than astronomy is about telescopes.”

— E. W. Dijkstra

# Contents

# Chapter 1

# Introduction

This chapter provides information about the context of the research presented in this thesis and defines the main research problem. Our approach is briefly discussed and an overview of this thesis is given.

## 1.1 Research Background

Software always depends on some other system for its functionality. Such a system can either be a hardware system or it can be another software system. Typically a system can be represented by a number of *layers*, which all provide functionality at different levels of abstraction. Any layer provides functionality to higher layers and conversely, a layer cannot operate without the functionality provided by the layers below. The bottom layer in layered architectures usually comprises hardware, while the top layer often consists of one or more software applications. To have an application operate properly on the hardware, intermediate layers such as an operating system are required.

As software is dependent on the hardware resources, it is preferable to have a way to verify whether the quality provided by the resources is appropriate for the software to operate properly. Testing is such a method, but can only be applied once the system has already been built. Unsatisfactory test results require code modifications or, in worst case, changes to the system architecture. Fixing problems afterward is costly and should be avoided [35].

### 1.1.1 System Definition

The Thales Naval Nederland (TNNL) Systems Definition business unit develops and designs Combat Management Systems (CMS). A CMS gets input from a number of sensors and provides output to actuators, as illustrated in Figure 1.1.

The systems are designed by dividing them into a number of functional components. For each component, functional flows are defined, describing the functionality offered by the component as a sequence of (high level) actions. From the functional flow, a decomposition into Computer Software Configuration Items (CSCIs) is derived, which all have their own responsibilities within the component. For the CSCIs in turn this process is repeated, producing even smaller, more specialized parts. Eventually the decomposition process results in parts that are small enough to be directly mapped onto software executables. The decomposition approach is typically a kind of *divide and conquer* approach, in which a complex system is split up into a number of less complex components [18].

The design of a system starts with the definition of functional requirements. These requirements have to describe *what* the system should do, that is, what its *functionality* is. In a layered architecture, the functionality a layer can expect from a lower layer is described

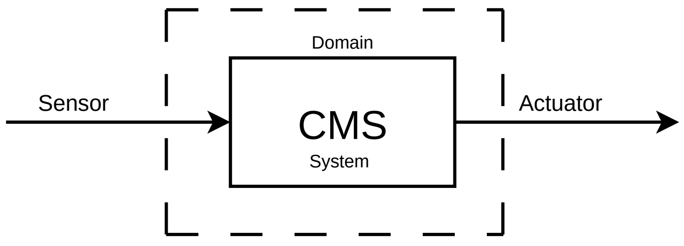


**Figure 1.1** — The Combat Management System gets input from sensors and provides output to actuators.

by the functional specification of the lower layer. Dependencies are only defined between a layer and its lower layers, because usually a layer only depends on the layer which is right below.

### 1.1.2 Non-functional Requirements

Besides the functional requirements, non-functional requirements can be defined for a system. Instead of stating what the system should do, they impose constraints on the freedom of software engineers as they design the system. Non-functional requirements limit the use of resources and set bounds on aspects of the software's quality, which is why they are sometimes known as *constraints* or *quality requirements* [1].

Non-functional requirements for components have a direct impact on their decomposition, because together the sub-components still need to satisfy those requirements. By setting non-functional requirements for a component, assumptions on the quality of service (QoS) provided by the component can be made by other components that depend on it. An example QoS property is the speed at which a component can process a certain amount of data. Quality requirements for components impose demands on the resources that are required by those components. Only if the resources are able to provide satisfactory QoS, the non-functional requirements for the components can be met. Reversely, whether or not the QoS can be considered good depends on the non-functional requirements of the component using the resource.

#### Standardization

For standardization of these constraints, the International Organization for Standardization (ISO) has defined the most common software quality characteristics in the ISO 9126 standard [16]. These characteristics and corresponding sub-characteristics are:

- **reliability**: maturity, fault tolerance, recoverability, availability, degradability
- **usability**: understandability, learnability, operability, explicitness, customisability, attractivity, clarity, helpfulness, user-friendliness
- **efficiency**: time behavior, resource behavior
- **maintainability**: analysability, changeability, stability, testability, manageability, reusability
- **portability**: adaptability, installability, conformance, replaceability

*Efficiency* is the important characteristic for our research, as it is the ISO term for performance related system characteristics. Both *time behavior* (attributes of software that bear on response and processing times and on throughput rates in performing its function) and *resource behavior* (attributes of software that bear on the amount of resource used and the duration of such use in performing its function) are important issues in the context of the research we present here [16]. However, we only focused on resource behavior. System behavior with respect to timing will be investigated in a subsequent study.

#### Verification

Not only the definition of non-functional requirements is important, but it is also vital that these requirements can be *verified* [18]. Usually, the verification is performed once the system, or part of it, has already been constructed. Failures which occur because of flaws in the high level design require greater changes to the product in order to fix the faults. We have therefore developed a method to eliminate such large design faults early in the software development process. Testing will still remain necessary, but the impact of discovered failures will not be as large as without the application of our method, resulting in a reduction of both cost and time.

#### Performance Engineering

The non-functional characteristics considered in our research are mainly concerned with performance. With *performance* of a component we denote the 'degree to which it accomplishes its designated functions within given constraints' [36]. Even though performance is critical for a lot of software systems nowadays, many fail to meet the performance objectives once they are constructed. It is therefore preferable to have some way to evaluate during, or at the end of an early development phase, whether the system still conforms to the conditions imposed at the start of that phase. Our method therefore ensures that the performance (non-functional) requirements can be *verified*, especially during early design phases.

Performance engineering is a systematic approach to developing systems that meet performance objectives. It begins early in the development lifecycle of a system, when changes to the structure of the system have the greatest impact. As Smith et al. aptly state: 'Performance cannot be retrofitted; it must be designed into software from the beginning. The "make it run, make it run right, make it run fast" approach is dangerous.' [35]

To support performance engineering of new systems, a three-step method is being developed at TNNL. The three steps each support verification of the system under development at different levels of abstraction. The steps are:

1. Components of the system are identified and organized. Budgets, in terms of resource utilization, are set for components.
2. High-level functional flows are identified in the system. Activity diagrams of flows are annotated with performance budgets.
3. The third step is a refinement of step 2. The annotated activity diagrams are further refined.

At every step a CMS is specified at a different level of abstraction. Performance characteristics are specified and verified at all three levels as well.

For the second step, a tool called DESIDE [14] has already been developed. The tool takes activity diagrams, annotated with resource demands and timing information, as input and can run a simulation to determine latencies and resource utilizations.

Before activity diagrams of high-level functional flows can be created, the high-level components need to be identified and organized. This is done in the first step and should give the engineer an initial idea about the resource utilization when a software system is deployed by a hardware system. Our research deals with the first step.

### 1.1.3 Performance Budgets

To evaluate whether the design of a system conforms to the performance requirements, we first required a way to express those performance requirements. A component imposes some load on the resources it uses and these resources have a limited capacity. To quantify resource demands of software and resource supplies of the hardware, these are referred to as *performance budgets* or simply *budgets*. Budgets of hardware and software are distinguished in the way that software *requires* resources and hardware *provides* resources. The corresponding budgets are called *required budget* and *provided budget* respectively.

### 1.1.4 Constraints

It is only possible to verify the conformance of a system to the specified performance requirements, when software can be related to hardware, or, required budgets can be related to provided budgets. To express these relations we provide the possibility to express *constraints* on the budgets.

Intuitively, the most basic constraint for any software – hardware relationship is that the required budget has to be less than the provided budget, because the software just cannot work without the required resources. More complex constraints, can however be expressed using our method as well. An example is the expression of a maximum load on the hardware. For instance as a percentage of the provided budget.

### 1.1.5 Validation and Verification

Model validation and verification are activities that proceed in parallel with the construction and evaluation of models [36]. Model *verification* is aimed at determining whether the model predictions are an accurate reflection of the software's performance. Model *validation* is concerned with determining whether the model accurately reflects the system it is supposed to describe [35].

Boehm has defined validation and verification in yet another way [2]:

**Verification** Are we building the model right?

**Validation** Are we building the right model?

The models created of a system naturally have to be validated, because they have to accurately reflect the system. Validation is a task of the engineer and is not supported by the results of our research. After successful validation, resulting in the conclusion that the model does properly reflect the system, the model has to be verified. In the context of this research, verification provides an answer to the question: 'Are the resource requirements that we have estimated reasonable?'. To support this decision, engineers can use the model and tool presented in this thesis.

## 1.2 Problem Statement

The main objective of the research presented in this thesis was to develop a method for specifying and verifying performance budgets in an early design phase for software intensive systems. Performance estimations were not made until the integration phase, which is too late.

Our research aims at moving performance estimations forward in the development process, by providing a tool for verifying performance budgets in an early phase. The method needs to be generic, such that a number of different systems can be specified with it. On the other hand, the method should not be too complex. According to the *Power Law of Data Standards*, the more systems that adopt a common standard, the simpler it must be [5].

The main problem has been divided into the following three sub-problems:

1. Definition of a method for the specification of performance budgets.
2. Creation of a system independent framework for the specification of constraints on performance budgets.

3. Development of an engine which calculates the resource utilization of a system.

As systems are specified using the Unified Modeling Language (UML) [25, 28], it is preferable that performance budgets can be specified in the UML as well. This ensures a good integration of system structure modeling and inclusion of performance properties. In the future, UML may be replaced by the Systems Modeling Language (SysML) [27], which has been developed for the modeling of complete systems rather than pure software systems. We did therefore take SysML into close consideration as well.

The use of UML for system definition and budget specification allows for the use of the Object Constraint Language (OCL) [23] to restrict the models. OCL is the obvious means of restricting UML models and did therefore seem to be an interesting candidate for the specification of constraints on budgets.

Verification of constraints results in an indication of resource usage by a system. The tool used within Thales for systems modeling is Rational Software Architect (RSA) [15], which is based on Eclipse [7]. It is preferred that specification of budgets and constraints, as well as verification of constraints are well integrated in RSA. As Eclipse, and thus RSA, can easily be extended by means of plug-ins, they are the preferred way of integrating the performance modeling functionality.

Whereas other studies on performance engineering [34, 35, 36] focus on the dynamic behavior of software systems, very little research has been conducted on static performance engineering in earlier phases of the development lifecycle.

The added value of this research is that it provides a way to statically verify models of systems with respect to resource demands in an early phase. Also, large and complex Excel sheets are no longer required to do the early performance calculations, because that is all covered by the study presented in this thesis.

## 1.3 Approach

Understanding the structure of the systems developed at Thales was not a very easy task. Available documentation consisted of an Excel sheet, which contains budget information for an example system. Consultation of ir. M.H.J. Glandrup and dr. ir. R.C. Scholte was required to clearly understand the structure of systems and the values and dependencies of budgets. No UML models of the systems were available to support this process of understanding. It is mainly from the Excel containing an example system sheet where the domain model of Chapter 6 has been derived. Initially, it could only be used to describe the example system, but now the model can be used to describe a wide range of systems.

For the implementation of the domain model, SysML did seem a good candidate, because it supports not only software, but hardware as well. A lack of tool support for SysML by RSA, compelled us to use the UML instead. The same problem existed for OCL, which did seem a good candidate to use for the expression of constraints. However, because of a lack of support for OCL by RSA we implemented our own, expression-based language to replace OCL.

## 1.4 Structure of this thesis

Chapter 2 introduces the basic concepts of modeling. Models and metamodels are described and some interesting architectural patterns are introduced.

Models of software are generally defined using the Unified Modeling Language (UML), which is described in Chapter 3. SysML, a new modeling language is described as well and a description is given about the Object Constraint Language in that chapter.

To illustrate various parts of this thesis, an example system is introduced in Chapter 4.

Chapter 5 corresponds to the first and second of the three requirements presented in the problem statement. It presents a model that captures the structural features of a system and provides the possibility to add budget information and constraints on the budgets.

The concrete implementation of the model in UML is presented in Chapter 6. This chapter corresponds to the third requirement of the problem statement.

A tool has been built which takes UML models with budget information and constraints as input and evaluates the constraints. A description of this tool is provided in Chapter 7.

Appendix A contains a precise description of the syntax of budget values and constraints. Finally, Appendix B can be used as a guideline to using the developed System Verifier Tool.

# Chapter 2

# Systems Modeling

This chapter provides an introduction to the concept of systems modeling as used in the context of this study. The terms *system*, *model* and *metamodel* are defined and explained.

## 2.1 Systems

In the field of computer science, people speak of systems in a number of contexts. Sometimes they refer to the hardware, sometimes to the software, to combinations of both and even to people. The International Council on Systems Engineering (INCOSE) [13] defines a system as "a combination of interacting elements organized to achieve one or more stated purposes". Kleppe et al. [17] define a system as "something that exists in reality". Both definitions can be applied to a variety of domains, for example to business process modeling (in case of which the business itself is the system). When talking about a system in the context of this research, we mainly refer to hardware and software. We therefore apply the following, more specific definition of a system, which is derived from the INCOSE definition:

**Definition 2.1** *A* system *is the collection of all hardware and software parts, organized to achieve one or more stated purposes.*

Hardware includes anything from specific components such as a single processor, to complete computer systems and federations of computer systems. Software includes all executable code which is or can be deployed on the hardware. When referring specifically to hardware or software, the terms *hardware system* and *software system* are used.

## 2.2 Models

Before a system is developed, it is first designed using models. In general, a model is an abstraction of something that exists in reality. Details are stripped away and only the elements that are relevant from a particular perspective remain [10].

A model can thus be seen as a simplification of reality, as it factors out any details that are irrelevant to the concepts which are presented in that model. In this sense, the model differs from the thing it models, because it contains less information. Furthermore, a model can be used as a blueprint to create something that does exist in reality, that is, a system [17].

From the very large number of diverse definitions of a model, we employ a definition which is based on the definition of a model as provided in [21]:

**Definition 2.2** *A* model *is the formal specification of the function and structure of a system.*

This definition can be extended by taking the definition of a model as provided by the metamodeling community into account: “An attempt at describing the world around us for a particular purpose” [33]. The word *purpose* is especially important here, as it would be pointless to include properties of the real world in the model that are not relevant to the intended goal of the model. For instance, the inclusion of climate information in the model of a software system defeats the object. This so-called abstraction is one of four requirements a good model should fulfill [18].

**Requirement 1** *A model should lead software engineers to gain insights about the system; they should be able to analyze the model to discover problems and other properties of it.*

In the context of this research, this requirement is the most important one for models. The aim of this research is to aid engineers in the design of a system and discover properties with respect to resource utilization. Besides having engineers analyze the model themselves, automated analysis of the model can provide even more insights to the engineers. The use of UML or SysML as a specification language for the model provides for all these requirements.

More about UML and SysML can be found in Chapter 3.

**Requirement 2** *The model should provide abstraction, such that not all details are always visible at once.*

UML and SysML allow the engineer to show as much detail as required. SysML even has a notion of views and supports the creation of models from different views in this way. Stakeholders involved in the design of a system focus on different concerns within that system. Hence different abstractions of a system are required in a single project. A *view* is an abstraction technique, which uses a selected set of architectural concepts and structuring rules, in order to focus on particular concerns [21].

Abstraction ensures that only the required details are visible to the engineer. Other, distracting details are kept out of the model, thus providing the engineer with a clearer picture of the system with respect to the concerns being treated.

**Requirement 3** *A model should be, as much as possible, understandable by all stakeholders such as clients and users, so they can participate in the development process.*

The third requirement has to be mainly fulfilled by the person who designs the model. He has to ensure that it remains clear enough for other stakeholders to understand. Clarity can be accomplished by defining multiple views on the model (see Requirement 2). The use of a well-known modeling language such as UML does also contribute to the understanding of the model, as many people are already familiar with its notation.

**Requirement 4** *A model should use a standard notation, so that it is interpreted in the same way by everybody who looks at it.*

According to Kleppe et al. [17], a model should be written in a well-defined language. The Unified Modeling Language (UML) [25, 28] and the Systems Modeling Language (SysML) [27] are both notations that have been standardized by the Object Management Group (OMG) [19] and are well-defined, which means they are suitable for automated interpretation by a computer. Using either of them fulfills this requirement.

## 2.3 Metamodels

Metamodels are closely related to models. Whereas models are abstractions of reality and aim at describing real-world phenomena, metamodels are abstractions of models and define constructs and rules needed to create those models. A metamodel can therefore be considered a language for the construction of models.

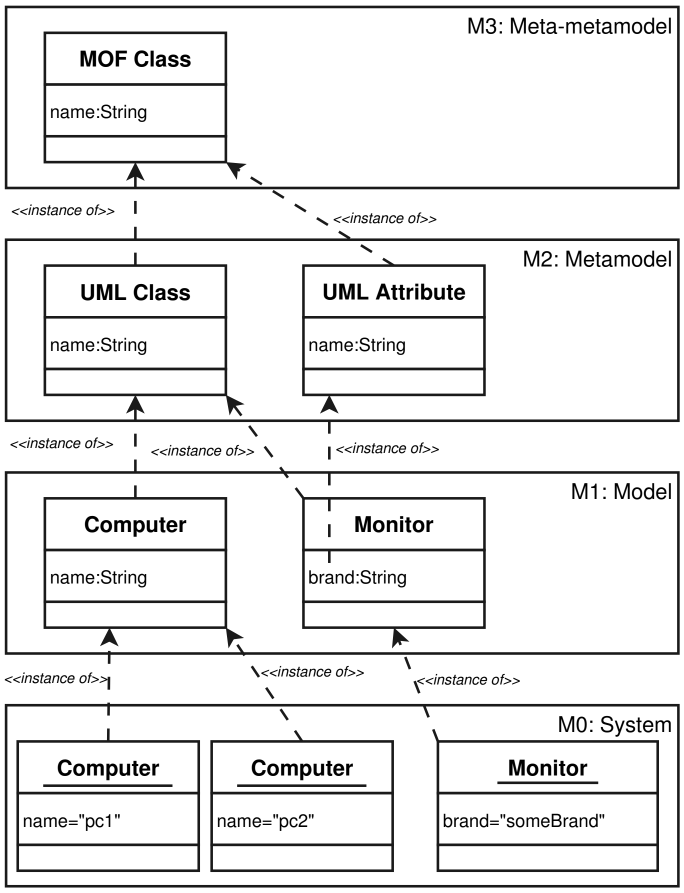


**Figure 2.1** — Overview of the OMG four-layer metamodel hierarchy

A model defines the elements that can exist in a system. If we have a model element *node*, then the instances of this model element, such as an *RT-node* exist in the system. A language, or, in this case, a metamodel defines what elements can exist in a model. For example, the UML language defines that the concepts of *class* and *package* can be used in a UML model [17]. In general three layers need always be taken into account when dealing with meta-layers to define languages [28]:

- the language specification, or the metamodel
- the user specification, or the model
- objects of the model

In theory it is possible to have an infinite number of metamodels, because every metamodel is written in a language which in turn can be described using some higher language. The language describing a metamodel is then referred to as a meta-metamodel. Instead of pursuing this further, we stick to the four-layer metamodel hierarchy of the OMG [28]. This metamodel hierarchy is illustrated in Figure 2.1. It shows an overview of the four layers M0 to M3, with examples of what can be present in those layers.

**M0** The lowest layer contains the instances of a running system, or UML objects. An example is the specific Computer with the name 'pc1'.

**M1** Layer M1 contains models, such as a UML model. The UML classes of this layer are the classifications of the instances at level M0. For example, the class ‘Computer’ is the classification of the two Computer instances ‘pc1’ and ‘pc2’ at level M0.

**M2** Layer M2 is a model of the model at level M1. Elements at this level are classifications of the elements at level M1. It contains the concepts needed to reason about concepts from layer M1. The model of this layer is called a *metamodel*. Every UML model at layer M1 is an instance of the UML metamodel, which resides at this layer.

**M3** This layer contains, just as the other layers, elements that define the elements at a lower layer. For the UML, this layer is called the Meta Object Facility (MOF) [26]. All modeling languages, including the UML, are instances of the MOF.

Instead of defining ever more layers, which is reasonably possible, the OMG defined that all elements of layer M3 must be defined as instances of concepts of layer M3 itself. As long as every element has a classifying metaelement through which metadata can be accessed, any model can be built and any system can be described [17]. Elements of the MOF can be described by elements of the MOF itself. Therefore, even without extra higher layers, all models can be described.

## 2.4 Architectural Patterns

Recurring design problems are best solved using well-known solutions from previous, similar design problems. This prevents the engineer from ‘reinventing the wheel’ every time a similar problem occurs. Whether or not the problem can actually be considered similar to a previous problem depends on a number of factors.

A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate [4].

This section presents the two architectural patterns that are relevant for the combat management systems of Thales. The Layers pattern and the Composite pattern are both incorporated in the model we created to describe CMS’s. The descriptions of both patterns are derived from Buschmann et al. ([4]) and described according to the method they provide. A pattern provides a solution to a general problem in a certain context. The description of a pattern according to Buschmann et al. thus defines the context, a general problem and the provided solution.

### 2.4.1 Layers

The *Layers* architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

**Context** A large system that requires decomposition.

**Problem** Designing a complex system which contains both low- and high level operations. Low level operations provide functionality to high level operations and, vice versa, the operations at a higher level rely on the operations at a lower level. A clear decomposition of the system is essential and interfaces between components need to be defined properly.

**Solution** Structure the system into a number of layers and place them on top of each other. Start with the lowest layer (layer 1) of abstraction and place higher layers on top of it (with layer N being the highest). Most of the services that layer J provides are composed of services provided by layer J-1. Layer J’s services may depend on other services in layer J.

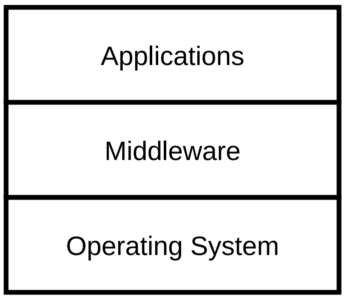


**Figure 2.2 —** An example of the layers pattern. This particular implementation consists of three layers, the OS, middleware and applications.

### 2.4.2 Composite

A *Composite* is an object designed as a composition of one or more similar objects that all exhibit similar functionality. The composite pattern allows a client to treat single components and collections of components identically. The pattern is illustrated in Figure 2.3.

**Context** A group of slightly different objects need to be treated in a similar fashion.

**Problem** Multiple objects are used in the same way and all have nearly identical code. Moreover, groups of objects are to be treated similarly to single instances.

**Solution** Create a *Composite* class, which is composed of abstract *Components*. The composite class inherits from this component class. Concrete implementations of the component class are the *Leaves*, which are not compositions of other components.

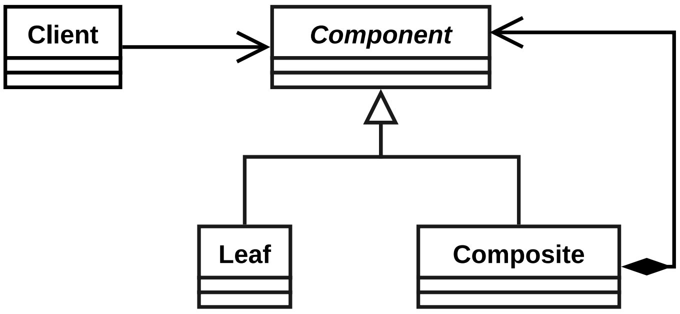


**Figure 2.3 —** The composite pattern

# Chapter 3

# UML, SysML and OCL

This chapter provides background information about the UML and SysML modeling languages and about the Object Constraint Language and its possibilities. Also, we present extension mechanisms which can be used to customize the UML.

## 3.1 UML Extension Mechanisms

The graphical modeling elements and relationships defined for UML diagrams are sometimes too limited for certain modeling tasks. It is then desirable that the precision of the UML models can be increased, to provide information for automated analysis or to specify the intent of a diagram more precisely. Moreover, it is often preferred that the UML can be extended to create new domain-specific modeling notations [32].

Currently, the two extension mechanisms that exist for UML 2.1 are profiling, also called a lightweight extension mechanism, and a heavyweight mechanism, as defined by the specification of the Meta Object Facility (MOF) [26, 30]. Both mechanisms provide extra possibilities to the user, but have rather different implications. Both extension mechanisms are described in Section 3.1. For the modeling of complete systems, the Systems Modeling Language (SysML) has been developed as a lightweight extension to the UML. SysML is described in Section 3.2.

### 3.1.1 Lightweight Extensions

The lightweight extension mechanism uses profiles to extend the UML. It consists of three main constructs: stereotypes, tagged values and constraints, of which the stereotypes are most important. It is called a *lightweight* extension mechanism, because it provides pure additions to the UML and does not change anything to the semantics of the metamodel elements, nor changes its structure nor adds new elements [30].

Whether or not to choose a lightweight metamodel extension technique depends on a number of factors. According to Desfray [6], a profile based technique should be chosen when:

- The domain is not subject to consensus, many variations and points of view exist.
- Many changes and evolutions may occur.
- The domain may be combined with other domains in an unpredictable way.
- Models defined in the domain may be interchanged with other domains.

#### Profiles

Since version 2.0 of the UML, a lightweight extension is generally defined as a UML profile (earlier versions of the UML did not contain the notion of a profile [28]). A profile is a special kind of package, which combines a set of stereotypes and tagged values. By applying the profile to a model (in the form of a package), the elements defined in the profile can be used in the model, thus providing an extension to the model.

#### Stereotypes

A stereotype defines how an existing metaclass may be extended. It is a metaclass itself, but can only be used in conjunction with the metaclasses it extends. This enables the use of specific terminology or notation for the extended metaclasses. Depending on whether the extension is required or not, the terminology or notation is either an addition or a replacement for the extended metaclass [28]. It is important to note that the semantics of the extended class cannot be changed by the stereotype.

Definition of a stereotype can be accomplished via the extension relation, as shown in Figure 3.1. A `{required}` constraint can be added to the extension relationship, denoting that every instance of the metaclass *must* be stereotyped.

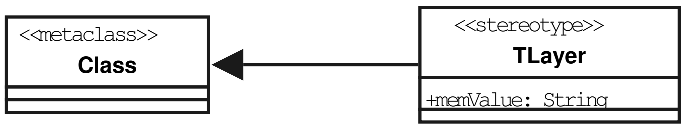


**Figure 3.1** — Definition of stereotype TLayer as an extension of the UML metaclass Class. The TLayer stereotype can thus be applied to any UML Class. The 'memValue' is a tagged value of the stereotype. Absence of the {required} constraint at the extension relationship specifies that application of the stereotype is not compulsory.

#### Tagged Values

Just like a class has attributes, a stereotype may have properties, which are referred to as tag definitions. Once the stereotype is applied to a model element, the values of these properties are referred to as tagged values.

In UML 1.3, tagged values could still extend a model element without the presence of a stereotype. Since UML 1.4 this behavior is deprecated and tagged values are only supposed to extend model elements as attributes of stereotypes [28].

An example of a tagged value is the `memValue` attribute of the TLayer stereotype in Figure 3.1.

#### Constraints

Besides tagged values, a stereotype may contain constraints with which the extended model can be restricted semantically. They are specified between braces { and } and can be expressed in any kind of language (e.g. English, OCL). A constraint is an assertion and is therefore not executable. An example of a predefined constraint is the `{required}` constraint on the extension relationship for stereotypes.

### 3.1.2 Heavyweight Extensions

Whereas lightweight extensions can only provide pure additions to the UML, heavyweight extensions can also change the semantics of the UML. This is done by explicitly adding new metaclasses and other metaconstructors which can introduce new behaviour [30]. This is in contrast with the lightweight stereotypes, which can only *extend* existing metaclasses and do nothing by themselves. A heavyweight extension technique should be applied when [6]:

- The domain is well defined and has a unique well accepted set of main concepts.
- A model realized under the domain is not subject to be transferred into other domains.
- There is no need to combine the domain with other domains.

Because of the fact that a heavyweight extension not just extends the language, but is also capable of changing it, it can be discussed whether the designation 'extension' is appropriate here. A heavyweight 'extension' can even change the complete syntax and semantics of a language and thus essentially defines a new language.

It is for this reason that it is very hard to combine multiple heavyweight extensions in a single model. Different languages do not combine very well. Furthermore, it is very hard for a tool to provide support for heavyweight extensions.

## 3.2 Systems Modeling Language (SysML)

The UML is widely used to create models of software systems. Both structural and behavioral properties of a system can be specified, using a variety of diagrams. For systems engineering however, the UML does not suffice. It has therefore been customized for systems engineering applications, which resulted in the OMG Systems Modeling Language (OMG SysML™), from now on referred to as SysML. Whereas UML unified the different modeling languages used in the software industry, systems engineers still use a wide range of modeling languages. SysML aims at unifying those languages [27].

SysML supports the specification, design, analysis, verification and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel and facilities. SysML reuses a subset of UML 2.1 and provides a number of extensions. The reuse of UML 2.1 elements allows software engineers and system engineers to easily collaborate on software-intensive systems. Whereas everything about systems used to be specified in a set of documents, SysML aims at providing a model centric approach, where the integrated models address multiple aspects of a system.

SysML extensions are defined using the following three extension mechanisms [27]:

- UML stereotypes define new modeling constructs by extending existing UML 2.1 constructs with new properties and constraints.
- UML diagram extensions define new diagram notations that supplement diagram notations reused from UML 2.1.
- Model libraries describe specialized model elements that are available for reuse.

No true metamodel changes are introduced by SysML, which can thus be considered a lightweight extension to UML 2.1.

The tree in Figure 3.2 shows the diagrams that have been taken from the UML, the modified UML diagrams and the new diagram types. Not all SysML diagrams are relevant for this research. The next sections describe the diagram elements and diagrams that are. They are mainly concerned with the definition of structural properties of a system (blocks and block diagrams) and with constraining parameter values (constraint blocks and parametric diagrams). Finally, Section 3.2.3 summarizes the allocation possibilities within SysML.

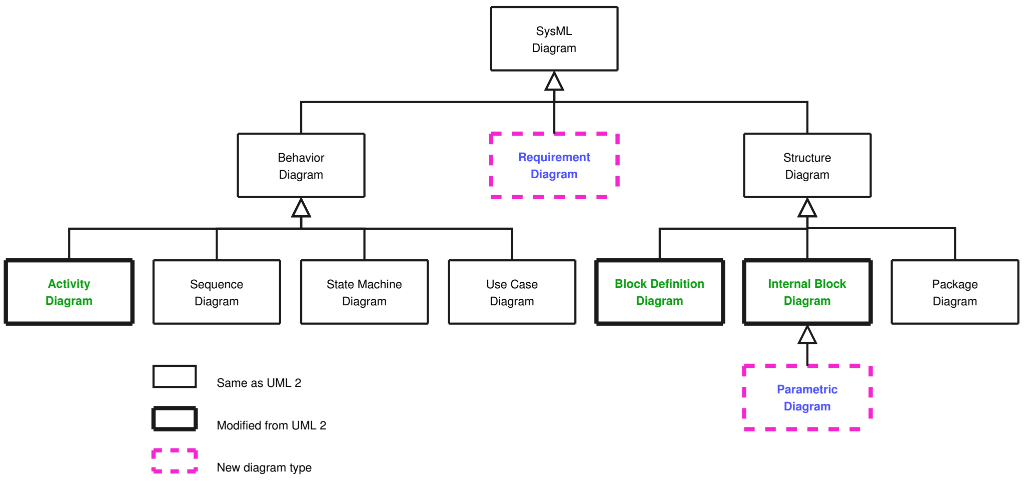


**Figure 3.2** — SysML diagram taxonomy [27]

### 3.2.1 Diagram Elements

This section contains a description of the SysML diagram elements which are relevant for this thesis.

#### Blocks

A Block is a basic structural model element, based on the UML class. It is a unifying concept which can be used to describe the structure of an element or a system. Blocks are meant to represent as many different kinds of elements as possible, such as hardware and software, but also data, procedures, facilities and even persons.

An example of a block is shown in Figure 3.3. A block can contain a number of compartments, such as constraints, operations, parts, references and values.[1] It is even possible to define custom compartments when necessary.

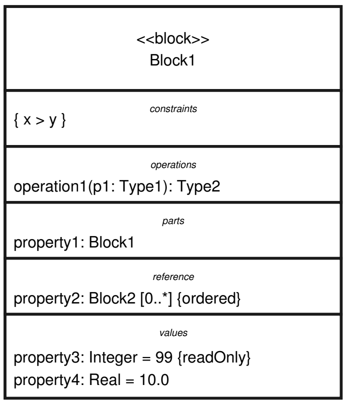


**Figure 3.3** — SysML block [27]

The *parts* compartment can be used to show the internal parts of the component represented by the block. Internal parts are also represented as blocks. They can either be

[1]More standard compartments are available. For a complete list please refer to [27].

added to the parts compartment or linked to the block by means of a composition relation. This is illustrated in Figure 3.4, where Block2 is a part of Block1. Using this convention, the structure of a complex system can be modeled in a hierarchical way.

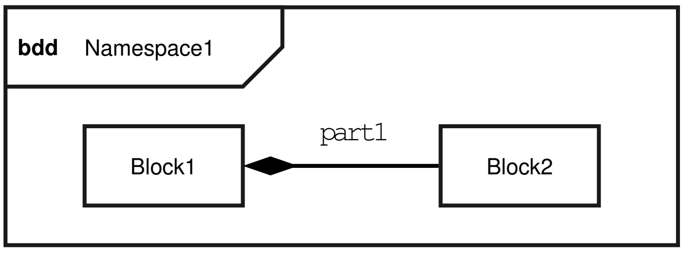


**Figure 3.4** — Block definition diagram (bdd) containing Block2 as a part of Block1 [27]

## Ports and flows

In SysML, flows between blocks can be specified by defining ports for those blocks and interconnecting them. A port is an interaction point between a block or part and its environment that is connected with other ports via connectors [27]. Whereas in the UML only flows of data could be specified, SysML allows for the definition of any kind of flows, such as the flow of liquids.

SysML defines two kinds of ports: standard ports and flow ports.

**Standard ports** define provided and required interfaces for the offered and expected services. Standard ports are also present in the UML (defined as ports) and are most used in the context of service-oriented architectures [27]. Standard ports are used for synchronous communication between blocks, though it is also possible to specify asynchronous communication by defining a service for signal reception. Figure 3.5 shows an example of a block with a standard port.

**Flow ports** are new to SysML and define the input and output items that may flow between a block and its environment [27]. The things that can flow in and out the FlowPort are defined by typing the FlowPort with those items. For an atomic FlowPort, this is only one type of item, but more types can be specified using a *flow specification*. FlowPorts are typically meant to be used for asynchronous interaction between blocks. Figure 3.6 shows blocks with example flow ports. The example shows three flowports *p1*, *p2* and *p3*.

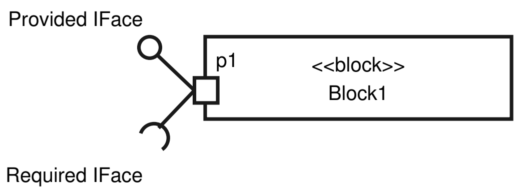


**Figure 3.5** — Standard Port with required and provided interfaces [27]

Whereas FlowPorts define what *can* flow through the ports, Item flows define what *does* flow through the ports. Consider a FlowPort which defines that it supports the flow of liquids. The Item flow could then specify that the liquid flowing through those FlowPorts is

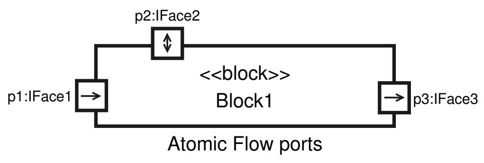


**Figure 3.6 —** Flow Ports with interfaces [27]

water. Item flows can be related to FlowPorts by means of allocation. An example of Item flow is shown in Figure 3.7. Block2 has an association *b1* with Block1 and Item flows from Block1 to Block2.

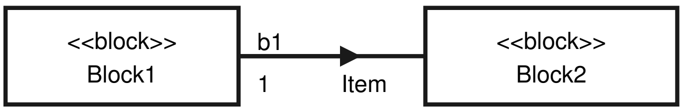


**Figure 3.7 —** Item flow specification [27]

ValueTypes

The properties of blocks need to be typed properly. For this purpose, new value types can be defined using a block definition diagram which contains blocks, stereotyped as `<<valueType>>`. Figure 3.8 shows an example of the definition of three new *ValueTypes*: temp, area and efficiency. All *ValueTypes* need to have a unit and a dimension specified. Specialized operations and properties can optionally be added to the available `operations` and `properties` compartments of the ValueType block.

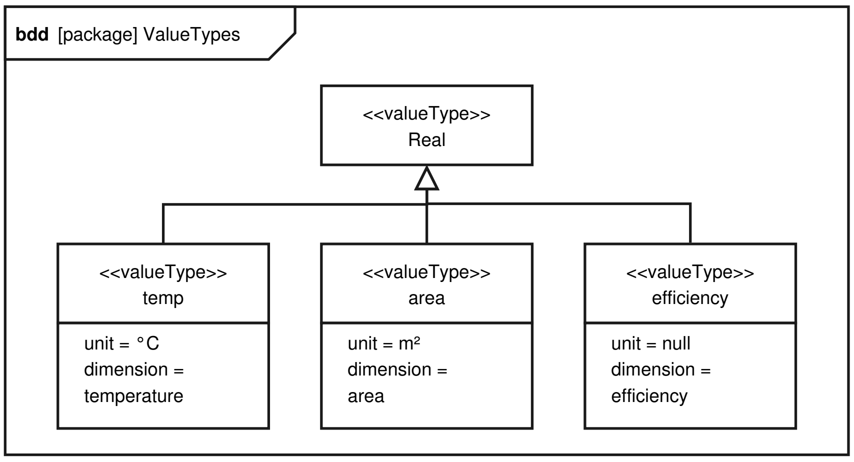


**Figure 3.8 —** ValueType definition (based on [11])

Constraint blocks

A constraint block is a special kind of block, stereotyped as `<<constraint>>`, which can be used to define generic constraints. It has two compartments, one defining the constraint itself and the other defining the parameters. Constraints can be specified using plain text, but also formal languages such as OCL or MathML can be used [27]. An example of a constraint block is shown in Figure 3.9.

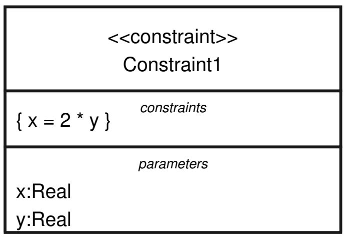


**Figure 3.9 —** Constraint block [27]

When the generic constraint is applied, it is represented by a rounded rectangle with a handle for each parameter. The handles can be used to bind the parameters to values, which can be done in the parametric diagram. The usage of a constraint block is illustrated in Figure 3.11, in which the property *length* of Block1 is bound to the parameter *x* of constraint block C1.

## 3.2.2 Diagrams

The diagram elements of the previous section can be applied in a number of SysML diagrams. The relevant ones are contained in this section.

#### Block Definition Diagram

From the taxonomy of Figure 3.2, it is clear that the block definition diagram (bdd) can be used to describe structural properties of a system. Figure 3.4 showed an example of a bdd, containing blocks in a part relationship.

#### Internal Block Diagram

The internal structure of a single block can be defined using an internal block diagram (ibd). An ibd is a special kind of block diagram, which is based on the UML composite structure diagram. Internal parts of a block are defined in the ibd, along with their interrelationships. A part is essentially a kind of block which resides in the `part` compartment of another block. Such a relation is already shown in Figure 3.4, where Block2 is a part of Block1.

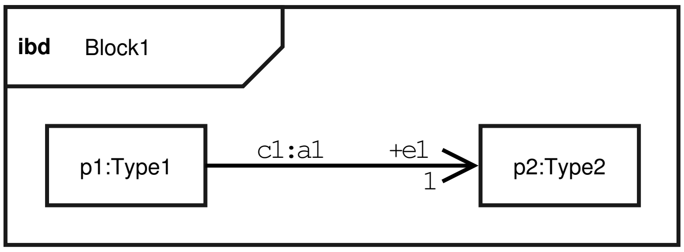


**Figure 3.10 —** Internal block diagram (ibd) [27]

The example in Figure 3.10 shows the internal structure of block *Block1*, which consists of two parts: *p1* and *p2*. The association relation shows that p1 has p2 associated to it.

### Parametric Diagram

Parameters of constraint blocks can be bound to values in a parametric diagram (par). These are the values specified in the `values` compartments of blocks. Figure 3.11 shows an example of a parametric diagram. The formal parameters `x` and `y` of constraint block `C1` are bound to the values of Block1 `length` and `width`.

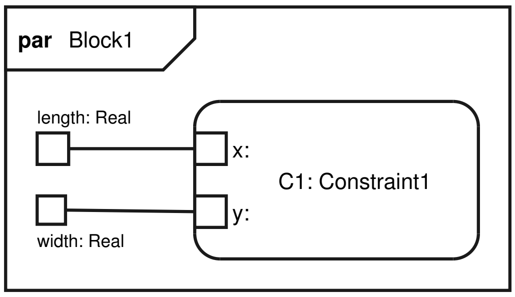


**Figure 3.11 —** Parametric diagram [27]. The parameters of constraint block C1 are bound to the values of Block1.

## 3.2.3 Allocation

SysML supports the allocation of diagram elements to other diagram elements and even the allocation of entire diagrams to diagram elements. This allows the engineer to make connections between different diagrams and construct a coherent model of a system. This section shows two different ways to specify allocations.

### Tabular format

The allocation of one diagram (element) to some other diagram element can be seen as a mapping between a number of diagrams and diagram elements. Using a table to represent this mapping is a rather straightforward way of doing this. An example is shown in Table 3.1. It shows the mapping of software components (components 1 to 3) to hardware nodes (node 1 to 3).

| type | name | end | relation | end | name | type |
|---|---|---|---|---|---|---|
| block | component 1 | from | allocate | to | node 1 | block |
| block | component 1 | from | allocate | to | node 3 | block |
| block | component 2 | from | allocate | to | node 1 | block |
| block | component 2 | from | allocate | to | node 2 | block |
| block | component 2 | from | allocate | to | node 3 | block |
| block | component 3 | from | allocate | to | node 2 | block |

**Table 3.1 —** Simple allocation using a tabular format.

The same can be accomplished using a matrix style, as shown in Table 3.2.

### Graphical format

Instead of providing a separate allocation specification, the *allocation* relationship can be used to define an allocation. Figure 3.12 shows this graphical representation.

| | Target | | |
|---|---|---|---|
| Source | node 1 | node 2 | node 3 |
| component 1 | allocate | | allocate |
| component 2 | allocate | allocate | allocate |
| component 3 | | allocate | |

**Table 3.2 —** Allocation using a matrix format.

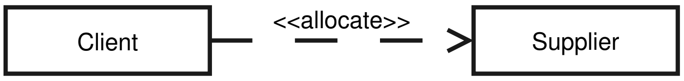


**Figure 3.12 —** Allocation relation [27]

Alternatively, allocation can be defined in the special predefined `<<allocatedFrom>>` and `<<allocatedTo>>` compartments of blocks.

## 3.3 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a formal language used to describe expressions on UML models. The expressions typically specify invariants that must hold for the system being modeled or queries over objects described in the model. OCL expressions do not have side effects when they are evaluated [23].

OCL is a pure specification language, not a programming language. Expressions can therefore only be evaluated, not executed, and control flow cannot be specified using OCL.

The main purposes where OCL can be used are the following [23]. Bold items are relevant for this research.

- **As a query language**
- **To specify invariants on classes and types in the class model**
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on Operations and Methods
- To describe Guards
- To specify target (sets) for messages and actions
- To specify constraints on operations
- **To specify derivation rules for attributes for any expression over a UML model.**

Using OCL as a query language allows the engineer to query the values of attributes. For instance, when class *Class1* has attributes *attr1* and *attr2*, the value of attr1 can be retrieved by the OCL expression:

```
Class1.attr1
```

Invariants can be used to constrain those values or relate values to each other. The value of *attr1* can be constrained (for instance, the value has to be less than 50) by the OCL expression:

```
Context Class1 inv:
    self.attr1 < 50
```

Finally, the initial and derived values of attributes can be specified using the `init` and `derive` language constructs respectively. For instance, to define that the value of attr2 is twice the value of attr1, the OCL expression reads:

```
Context Class1::attr1: Integer
    init: attr2 * 2
```

## 3.4 Evaluation

The models we present in Chapters 5 and 6 aid engineers in the modeling of domain concepts. An extension mechanism can provide the additions required to express these domain concepts. The choice of an extension mechanism is heavily dependent on the support provided by tools.

SysML seems to be the perfect candidate for defining the models of this research. Especially because of the steps following this research (see Section 1.1.2, specification of functional flows, etc.) SysML can very well be applied. The models created in the first of the three steps can easily be extended to suit the following steps. Hence, the model created in the first step can be ported to the second and third and only this single model is required to describe all levels of detail.

The tool used within Thales to create UML models, Rational Software Architect (RSA [15]) does support the lightweight extension mechanism, but provides no support at all for heavyweight extensions. Support for SysML, which is defined as a UML profile, is provided as an RSA plugin by a third-party company, called EmbeddedPlus Engineering [9]. Experiments with this plugin proved that its performance did not meet the expectations. Mainly because SysML is still in an early stadium, tool support for the language is poor. Despite its promising features, SysML is therefore not used as a modeling language in this research.

Due to the facts that heavyweight extensions do not combine easily and that tools generally do not provide support for heavyweight extensions, the lightweight extension mechanism best fits our situation. Support for SysML is still poor, so a custom profile has been created (presented in Chapter 6) which provides all the support to engineers in creating the models described in this thesis.

OCL seems a good candidate for the specification of attribute values and the definition of constraints on UML models. However, just as with SysML, the support for OCL is poor in RSA. Constraints can only be evaluated for elements of the metamodel (see level M1 of the OMG four-layer metamodel hierarchy in Figure 2.1), but not for the user model (layer M1).

Other OCL implementations (such as OCTOPUS [20]) do not combine well with RSA, because of the data structures they use to store UML models. Using such a different implementation would require the UML models produced by RSA to be translated into another structures. This, however, would take far too much time to complete and would not be a valuable contribution to this research.

# Chapter 4

# Example System

This chapter introduces an example Combat Management System and its most important parts. This example is derived from the example Excel sheet and used throughout the thesis for illustration purposes.

## 4.1 Structure

The Combat Management System, or CMS, consists of a hardware part and a software part. The hardware provides a number of resources such as processing power (CPU) and memory to the software. The software on the other hand requires these resources to properly perform its functions.

### 4.1.1 Hardware

The hardware system (the hardware) is composed of a number of subsystems, called nodes, which are interconnected by a network. This particular CMS consists of 8 console nodes, 1 real-time (RT) server and 1 non-real-time (NRT) server.

**Console** A console is used to present information to an operator and allows the operator to interact with the CMS. The execution of Human Computer Interaction (HCI) applications is an important task of the console nodes. Besides HCI applications, other applications which provide processing functionality are also executed by console nodes.

**RT server** A server dedicated to running real-time applications.

**NRT server** A server for the processing of non real-time applications. The NRT server contains a database for data storage and deploys Commercial-Off-The-Shelf (COTS) software.

### 4.1.2 Software

A software system (the software) is organized using a layered architecture [4] and, as usual for the software systems built by Thales, consists of three layers:

**Computer Software Configuration Items** The applications in the top layer are the Computer Software Configuration Items (CSCIs). A single CSCI has a certain task, such as processing radar data or providing information to an operator. CSCIs are loosely coupled and share data by means of the publish-subscribe mechanism of the SPLICE middleware.

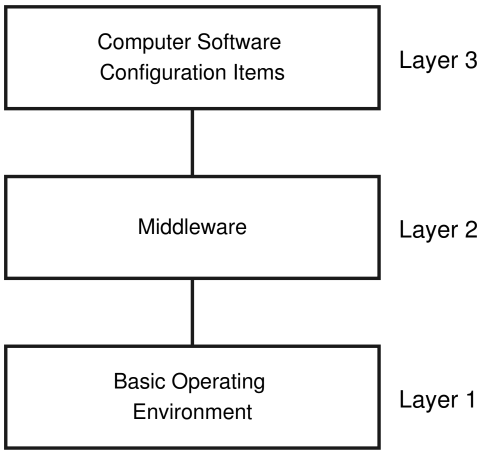


**Figure 4.1** — The three layers of the example CMS software system

**Middleware** The middleware for the CMS is called SPLICE-DDS and is based on the OMG's Data Distribution Service for Real-Time Systems Specification [22]. The SPLICE architecture is characterized by autonomous applications with minimal dependencies where function and interaction are clearly separated and SPLICE agents act as real-time information brokers [12].

To support both legacy and recent applications, a special software library called S142 (SPLICE 1 for 2) is included in the middleware. S142 provides SPLICE-1 legacy support to applications which cannot make use of SPLICE-2 (SPLICE-DDS) directly. S142 is a vital part of this example CMS.

**Basic Operating Environment** The Basic Operating Environment (BOE) is present on almost every hardware node in the system. It includes a number of components, such as the Operating System (OS) and other supporting applications.

Each layer is composed of a number of components, which together make up the functionality of that layer. Depending on the type of node, components are or are not present on that node. The components of the BOE are almost always available on every node deploying Thales' own software, because of their basic, supporting functions. The CSCIs layer on a node only contains the components necessary for that node.

## 4.2 Budgets

The Console nodes of the hardware system do all provide the same types and amounts of resources. The example CMS consists of eight Console nodes, thus providing eight times the amount of resources of one Console node. The RT node has a different purpose and provides a different amount of resources.

As described above, the software system of the example CMS consists of three layers. The components of each layer have their own required budgets, which consists of a fixed part and a part that depends on other budgets. For example, some components of the CSCIs layer can use the SPLICE-DDS middleware directly. Other, legacy components require the functionality of the S142 component of the middleware layer to make use of SPLICE-DDS. Depending on the amount of data a legacy component exchanges with SPLICE-DDS, the overhead introduced by the S142 component will increase. The budget of a component in the CSCIs layer is therefore sometimes dependent on the budget of a middleware component.

## 4.3 Allocation

Not all software parts are allocated to all hardware nodes. The software parts that impose real-time constraints on the hardware are deployed by the RT server. On the other hand, the software parts that are deployed by the NRT server do not require the functionality provided by the complete BOE and thus does not deploy the BOE, which *is* present on all other nodes of the CMS.

Due to the early stage in the development process, the exact allocation of the CSCIs to the hardware is not known. Estimations are however made. For instance, engineers have estimated that 25% of all CSCIs of the example system will run on only 40% of the available nodes:

- 20% of the CSCIs is allocated to 100% of the nodes
- 25% of the CSCIs is allocated to 40% of the nodes
- 55% of the CSCIs is allocated to 20% of the nodes

Even though not specified for each CSCI individually, they are all allocated to the hardware. Some nodes deploy all CSCIs, while others only deploy a subset.

# Chapter 5

# Domain Viewpoint

This chapter describes the domain viewpoint, which is a conceptual model of a system. The contents of this chapter corresponds to the first and second requirement of the problem statement of Section 1.2 and defines a method to specify performance budgets and constraints. The main concepts of the structure of hardware and software, budgets, allocation and constraints are discussed in order. We derived the model presented in this chapter from an Excel sheet describing the example system of Chapter 4.

## 5.1 Introduction

As described in Chapter 4, the example CMS consists of a hardware part and a software part. Generally, any CMS consists of both hardware and software. In this chapter, a model is created by which both the hardware and the software can be described. Furthermore, the model describes how to link the software to the hardware, to denote which software parts are allocated to which hardware parts. Finally, restrictions are imposed on the provided and required budgets, thereby relating them to each other.

We derived the model presented in this chapter from the Excel sheet containing the example system and modified it to describe not only that system, but combat management systems in general.

Figure 5.1 shows an overview of the domain. A CMS consists of a software system and a hardware system, which both have a hierarchical structure. Any part of the software system can be allocated to any part of the hardware system, expressing that the software part is executed by the hardware part. All software parts can have required budgets, which are related to the provided budgets of the hardware in constraints. The constraints reside on the allocation relationships.

The following sections discuss the structure of the hardware and software, the definition of budgets, allocation relationships and constraints. This results in the model of the structure of a CMS, which includes the possibility to specify budgets. Budgets are related by defining allocation relationships and adding constraints. The complete model satisfies the first and second part of the problem statement.

## 5.2 Hardware

Several types of nodes may exist in a single hardware system. For example, the CMS example system in Chapter 4 consists nodes of three types. In general, a hardware system can consist of an arbitrary number of nodes which all have a certain type.

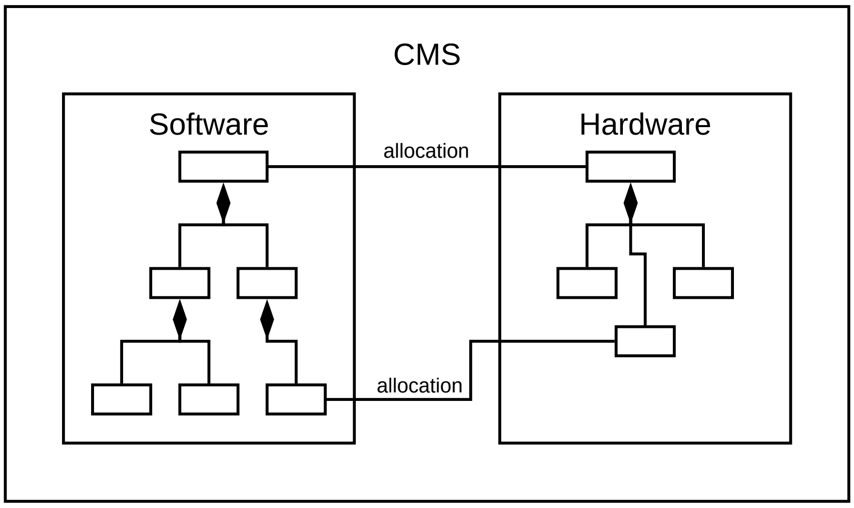


**Figure 5.1 —** Domain overview. A CMS consists of software, hardware and allocation relationships relating them.

### 5.2.1 Structural Decomposition

A hardware system is organized in a hierarchical way. In general, two different model elements can be distinguished for any hardware system: a *Node* and a top-level *HardwareSystem*. The HardwareSystem is defined by the nodes it consists of. This is illustrated in Figure 5.2.

Changing the number of nodes in a hardware system results in a new configuration. For instance, the example CMS consists of eight Console nodes, one RT node and one NRT node. A different hardware system can consist of not eight, but twelve Console nodes. The functionality of such a configuration is the same as the example CMS, but provides more resources.

### 5.2.2 Domain Model

The domain model of the hardware system is shown in Figure 5.2. A composition relationship defines that all nodes together make up the hardware system. Budget information can be added to the Nodes as well as to the top-level HardwareSystem.

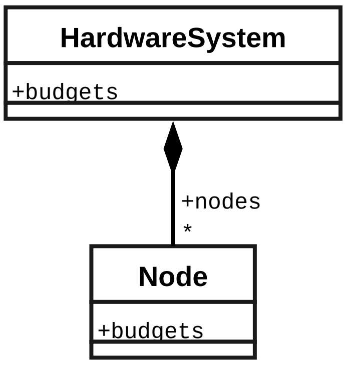


**Figure 5.2 —** Hardware system domain model

## 5.3 Software

Besides the hardware, software is the other main part of a system. The model describes how a software system is decomposed in a hierarchical way.

### 5.3.1 Structural Decomposition

The highest conceptual level is the software system itself, which, when completely decomposed, consists of a number of components. Each component has some predefined function or service to fulfill and is responsible for the execution of that specific task. Even though some components are completely independent of other components to fulfill their tasks, most rely on the services of other components. This behavior is reflected in the structural decomposition of a software system in layers.

In general, three model elements can be distinguished in the domain model of the software system: the top-level *SoftwareSystem*, the *Layer* and the *Component*. This is illustrated in Figure 5.3.

### 5.3.2 Domain Model

The layers pattern and the composite pattern which were described in Section 2.4 are applied to the software system domain model. Recall that in the example CMS three layers have been identified (the BOE, the middleware and CSCIs). Because a layered pattern can be identified in any CMS, it is included in the domain model. The layers together make up the SoftwareSystem, as illustrated by the composition relationship in Figure 5.3.

A layer groups a number of components. Figure 5.3 illustrates this, by the composition relationship between the Layer model element and the Component model element. A single component fulfills some task, which can be divided into sub-tasks, performed by sub-components. The composite pattern is applied to components and allows them to be specified at any level of decomposition.

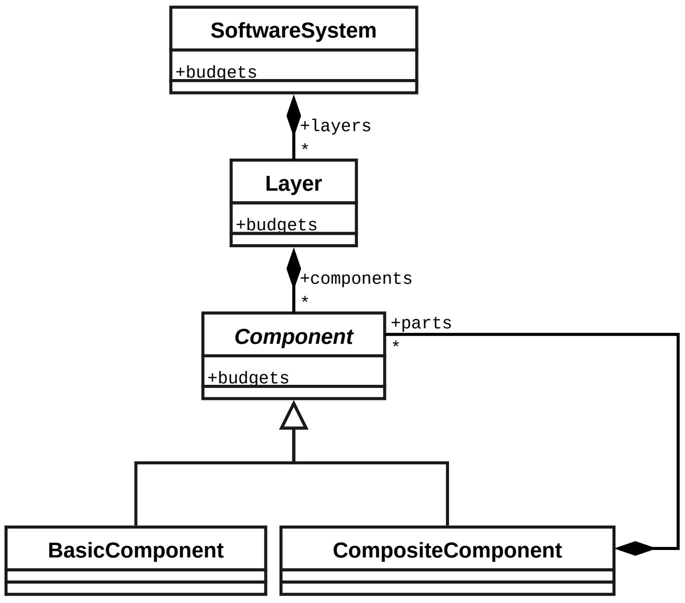


**Figure 5.3 —** Software system domain model

Essentially, real-world instances only exist for the BasicComponent model elements. The functionality a component provides is realized by a piece of code. The complete collection of all code is called the software system, which itself is merely a mental grouping concept. The software system only exists by grace of the components of which it is composed.

A layer is even more special. Just as the software system it is a grouping concept, because no real-world instance of a layer ever exists. A layer groups a number of components providing functionality at a similar level of abstraction and thus provides a way to hide implementation details.

In reality, the existence of layers can only be derived from the interactions between components. The boundaries of layers are formed by interfaces, defining the communication possibilities for components. A reflection of a real-world system should therefore *not* contain layer entities.

For modeling purposes however, the layer is important to include in the model. Systems are explicitly designed in layers, because of the many advantages a layered architecture possesses [4]. So even though a layer does not explicitly exist in a system, it does exist in the mental model of the engineer. It is therefore a logical consequence to include the layer in the mental model of Figure 5.3. Furthermore, engineers need to add budget information to entire layers. This is only possible if the layer is included in the domain model of the system.

## 5.4 Budgets

The amount of resources required by software depends on a number of factors, such as the type of resource and the implementation of the software. To express the amounts of resources demanded by a software program and offered by a hardware node, Thales employs the following definition of a *budget*.

**Definition 5.1** *A* budget *is the representation of an amount of resources.*

Budget information is specified for both hardware and software. The provided and required budgets are only distinguished by by the parts of the system (hardware or software) on which they reside. Budgets can be defined for all parts of the system. The model element containing the budget is referred to as the *owner* of the budget.

### 5.4.1 Types

Different types of resources demand different types of budgets to be described. The budget value of a specific CPU in a computer system will have a completely different meaning than the budget of the memory in that same computer system. To make the distinction between types of resources, *type* information is added to the budgets.

For each type of resource, a corresponding budget type is required. Four basic budget types are used often and made available by default for all model elements. The text between brackets is to be used when referring to the budget. For instance, a reference to the memory budget of the model element BOE is: *BOE.mem*.

- CPU (cpu)
- Memory (mem)
- Storage (stor)
- Networking (net)

### CPU

The processing power required by software is highly dependent on the type of CPU, the used compiler and other factors for which it was designed. A good representation of the CPU budget should abstract away from these dependencies and define a universal measure for processing power. Unfortunately, this is very difficult to realize. Two example representations of processing power are shown below.

**MIPS** MIPS is an acronym for Million Instructions Per Second. It defines the number of instructions a CPU can execute within a time unit of one second (e.g. a MIPS value of 2 means the CPU can execute $2 \times 10^6$ instructions per second).

**SPECint** The Standard Performance Evaluation Corporation (SPEC) [37] is a non-profit organization whose goal is to establish, maintain and endorse a standardized set of relevant benchmarks for computer systems. SPECint is one of those benchmarks, aimed at measuring and comparing compute-intensive integer performance. By avoiding loads on other components in the computer system such as I/O and networking, the benchmark is as accurate as possible.

The SPEC maintains a list of benchmark results for a variety of computer systems and test programs (e.g. data compression with gzip and compilation with gcc). These results, expressed as a single value per benchmark, can be used to derive an indication of the performance of some new system, or a benchmark can be performed on the system to get a more accurate result.

The best way to represent processing power is by means of the SPECint benchmark, as it allows a comparison of one software system on different types of hardware. However, the option to use MIPS needs to remain available too. SPECint benchmarks are not always available for just any kind of system and performing a benchmark is a costly and time consuming activity. Moreover, a running system is required to do the actual benchmark, rendering it unsuitable for systems under development. The speed in MIPS is generally available for most CPU types, making it a more suitable measurement in some cases. It is easier to make an estimation of processing power in terms of MIPS, which is why both approaches remain available.

### Memory

Data stored in memory is transient, so after the application has halted it is no longer available. Memory demands are measured in terms of the size of data. General data types such as an integer can be assigned different sizes, depending on the compiler used to translate the program. Despite this dependency, memory usage is measured in Bytes. Multiples of 1024 (i.e KByte, MByte and GByte) are used to define large data sizes.

### Storage

Data storage is measured similarly to memory usage. In contrast to memory, data storage makes the data persistent, so it does remain available after the program has halted. Depending on the way the data is stored (for example directly to a disk or in a database), the storage size of the data can vary, because some storage ways introduce more overhead than others.

Data storage is represented in the same way as memory; in Bytes. Again, to represent large amounts of data, the units KByte, MByte and GByte can be used.

### Networking

Components can communicate with each other using a network connection. A connection can only process a certain amount of data, measured in bits per second (bps). Provided and required budgets are also measured in bits per second, but since values tend to get very large using only this unit, the *Kbps*, *Mbps* and *Gbps* units are also available. Just as with memory and storage, these units differ a factor 1024 from each other.

### 5.4.2 Values

Budget values are not always known a priori. Application of the layers architectural pattern implies the existence of dependencies between components. These dependencies exist at least between components in adjacent layers, but dependencies between components within the same layer are also a possibility. Consequently, budgets of independent components usually have fixed values, whereas budgets of dependent components can have their own dependencies. Budgets of composite components (and of layers and a software system) are, if not explicitly defined, dependent on the budgets of the children.

**Fixed** Some budgets simply have a value that does not change under the influence of other budgets or other factors. Such a budget has a fixed value for the lifetime of the owner of the budget.

**Dependent** Some budgets do not have a fixed value, but have a value that changes depending on the values of other budgets. The value of such a budget is not known until the value of the budget it depends on is known. Determining the value of a budget might require some calculations to be made.

**Sum** The structural hierarchies of both hardware and software systems introduce yet another way to derive the value of a budget. For high-level model elements in a decomposition hierarchy (HardwareSystem, SoftwareSystem, Layer and CompositeComponents), budget values depend on the values of the contained model elements (the children).

If no value has been defined for a budget with some specific name, then its value is defined as the *sum* of the budgets with that name of the owner's children. In the case where both values are present, the largest of the two is taken as *the* value. The engineer thus has the freedom to first create a high-level design and extend and refine it afterwards.

## 5.5 Allocation

After both the models of the software and the hardware have been created, they need to be connected. All parts of the software system have to be allocated to the hardware nodes on which they can operate. This is done by the allocation specification. For each combination of a software configuration and a hardware configuration, an allocation specification needs to be available to connect the both.

To define that a software part runs on some hardware node, an *allocation* relationship is created between the two. Besides this main purpose, an allocation relationships defines which software and hardware parts of a system can take part in the verification process as well. Whether or not they actually do, depends on the constraints (Section 5.6). Allocating a software model element to a hardware model element means that the software element is deployed by the hardware element and, subsequently, that the software element uses the resources provided by that hardware element.

### 5.5.1 Simple Allocation

Software parts (software system, layer, component) can all be allocated to hardware parts (hardware system, node) in any possible combination. Including a composite part (i.e. software system, layer, hardware system or composite component) in an allocation relation means that in fact all *contained* parts take place in the allocation. Allocating a layer to a node thus means allocating all components in that layer to the node. Figure 5.4 shows the allocation of BOE to all Console nodes. BOE consists of a number of components, which are all allocated to all Console nodes.

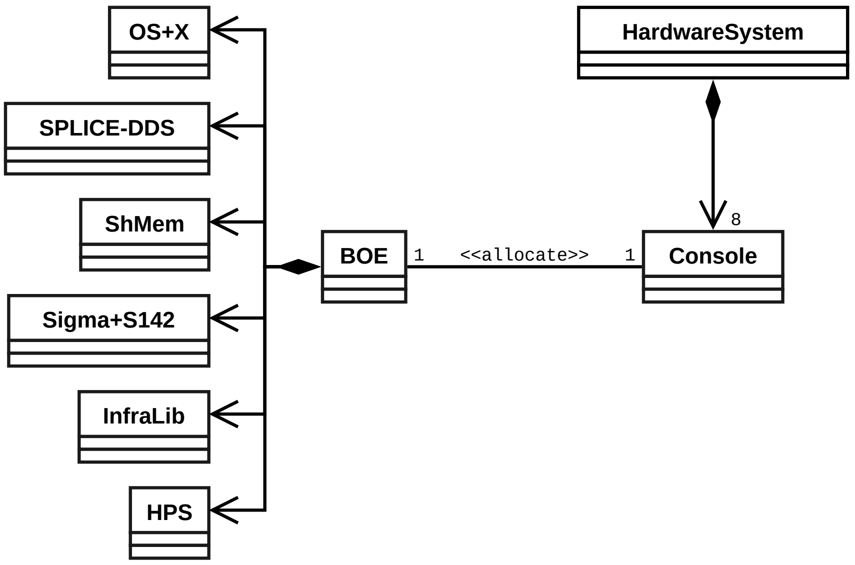


**Figure 5.4** — Example allocation of the BOE layer, consisting of components to all Console nodes of the example CMS.

Whereas the BOE of Figure 5.4 is only allocated to the Console nodes, it can easily be allocated to all nodes of the hardware system, by allocating it to the HardwareSystem itself. This is illustrated in Figure 5.5, where the layer *BOE* is allocated to all nodes of the example CMS hardware system, consisting of eight Console nodes, one RT and one NRT node.

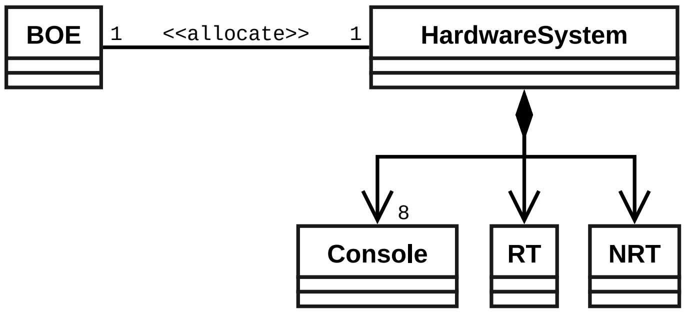


**Figure 5.5** — Allocation of one layer (BOE) to all nodes of the HardwareSystem.

Any part of the software system can be allocated to any part of the hardware system. Allocating a composite element (such as a layer) to a node means that all its components run on that node. Also, allocating a software part to the top-level HardwareSystem node means that the software part runs on all nodes of the hardware.

### 5.5.2 Complex Allocation

Instead of allocating a complete layer to a set of nodes, it is also possible to allocate a part of the layer to some of the nodes of the hardware system. Complex allocation allows

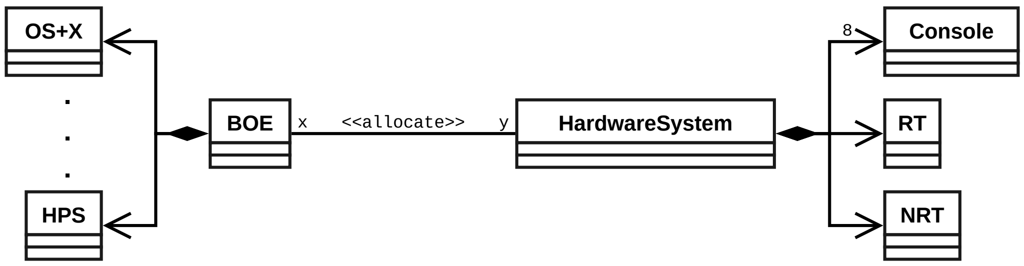


**Figure 5.6 —** Allocation of ratio $x$ of layer BOE to ratio $y$ of the HardwareSystem.

the allocation of all parts of a composite model element, either at the software or at the hardware side, without having to specify the allocation for each of the parts individually.

A complex allocation relationship is illustrated in Figure 5.6. The values of $x$ and $y$ are called *ratios*.

Whereas the UML only allows integer values for the multiplicities at the association relationships, ratios can have all values $> 0$ and $\leq 1$. Even though RSA claims to implement the UML, allowing real numbers as multiplicities is a deviation from the UML standard.

The interpretation of the allocation relation of Figure 5.6 is that ratio $x$ of the layer BOE is allocated to ratio $y$ of the hardware system. In practice, it is not possible to exactly represent an exact ratio of a layer or of a hardware system. Instead, the values of $x$ and $y$ are used to calculate the contribution of the required budget in the allocation relationship to the final required budget of the software system.

Suppose the required memory budget of each child components of the BOE is 100 MByte and all 10 nodes of the hardware system have a provided budget of 1 GByte. Furthermore, ratio $x$ has the value $0.75$ and ratio $y$ has the value $0.5$. This has to be interpreted as: *75% of the budget of BOE is allocated to 50% of the nodes of the hardware system.*

Generally, the contribution of the required budget of any software part can be calculated as follows, with $x$ being the ratio at the software side of the allocation relationship and $y$ the ratio at the hardware side:

$$\text{contribution} = \begin{cases} \text{budget SW} * \#\text{children of HW} * x * y & \text{if HW has children} \\ \text{budget SW} * x * y & \text{if HW has no children} \end{cases}$$

Excluding the budget of the software results in a factor that can be calculated for each allocation relationship, independent of the required budget of the software part:

$$\text{factor} = \begin{cases} \#\text{children of HW} * x * y & \text{if HW has children} \\ x * y & \text{if HW has no children} \end{cases}$$

Using this kind of allocation specification, it is also possible to specify that some software parts have to be allocated to a set of nodes, while other software parts need to be allocated to other nodes. This is illustrated in Figure 5.7. One allocation relationship defines ratios $x_1$ and $y_1$, while the other defines $x_2$ and $y_2$. In case of $n$ allocation relationships, the factor is calculated as follows:

$$\text{factor} = \begin{cases} \#\text{children of HW} * \sum_{i=1}^{n} x_i * y_i & \text{if HW has children} \\ \sum_{i=1}^{n} x_i * y_i & \text{if HW has no children} \end{cases}$$

For $x_1 = 0.75$, $y_1 = 0.5$, $x_2 = 0.25$, $y_2 = 1.0$ and a required budget of 100 MByte per component of the BOE, the contribution of the software is calculated:

$$\begin{aligned}\text{contribution} &= 6 * 100 \text{ MByte} * \text{factor}\\ &= 600 \text{ MByte} * (10 * ((0.75 * 0.5) + (0.25 * 1.0)))\\ &= 600 \text{ MByte} * 6.25\\ &= 3750 \text{ MByte}\end{aligned}$$

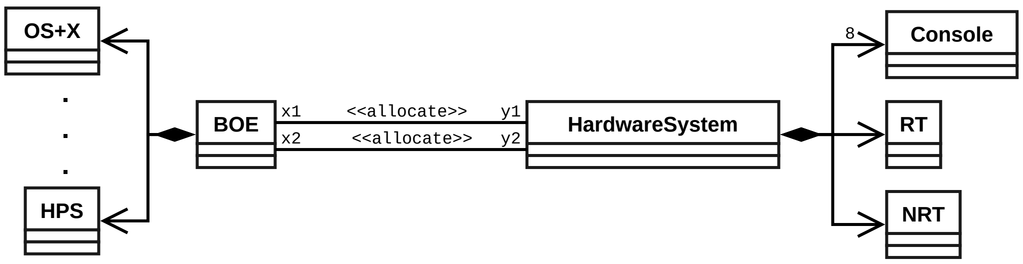


**Figure 5.7 —** Multiple allocation relationships between two model elements.

## 5.6 Constraints

After the allocation has been defined for a system, budgets can be restricted by means of constraints. A constraint relates two or more budgets, usually at least one required and one provided budget, to each other. This way, budgets of software and hardware can be compared. Constraints are specified as boolean expressions and are evaluated automatically. In the context of Thales' combat management systems, we define a constraint as follows:

**Definition 5.2** *A constraint is a rule, that relates budgets of software to budgets of hardware and can automatically be verified. Evaluation of the constraint always results in either the outcome true or false.*

Because constraints are always defined over the boundary between software and hardware, they depend on the allocation specification for the system. Only budgets of model elements included in an allocation relationship can be related in a constraint. The allocation specification thus defines the possibilities for defining constraints.

Constraints are always specified as boolean expressions, which contain references to budgets. A basic example is shown below. The constraint defines that the memory budget of the BOE has to be less than 25% of the memory budget of the hardware system.

$$\text{BOE.mem} < 0.25 * \text{HardwareSystem.mem}$$

Evaluation of this constraint takes the allocation specification into account and uses it to determine the value of BOE.mem. The calculation at the end of Section 5.5.2 shows how the value of BOE.mem is determined with the provided specification allocation. The resulting value of $3750$ MByte is only relevant when applied in a constraint on the particular allocation relationship. In all other cases, the value is simply the original budget of the BOE: $6 * 100 = 600$ MByte.

For this example, the result is:

$$\begin{aligned}\text{result} &= \text{BOE.mem} < 0.25 * \text{HardwareSystem.mem} \\ &= 3750 \text{ MByte} < 0.25 * (10 * 1 \text{ GByte}) \\ &= 3750 \text{ MByte} < 2.5 \text{ GByte} \\ &= 3750 \text{ MByte} < 2.5 * 1024 \text{ MByte} \\ &= 3750 \text{ MByte} < 2560 \text{ MByte} \\ &= \text{false}\end{aligned}$$

With the current budget values of 100 MByte per BOE component, 1 GByte per hardware node and the two allocation relationships, the memory budget of the BOE is *not* less than a quarter of the total budget of the hardware system.

## 5.7 Discussion

A distinction is made between the structure of hardware and software. A hardware system only consists of two decomposition levels; the complete hardware system and its nodes. The domain model is based on the example CMS, of which the hardware is simply specified as a number of nodes. A more fine-grained decomposition is not required for performance verification at this level, for which only the number and types of nodes are necessary.

For a software system, an arbitrary number of decomposition levels can be specified. The high decomposition levels are included as specific domain elements (the layer and component), but further decomposition is accomplished by the application of the composite pattern. Layers have a special function in the structural hierarchy of a software system, because they group components depending on he level of abstraction of their functionality.

Resource amounts are defined in terms of budgets. A budget represents an amount of a resource of a specific type and has a unit. When related to software, a budget is required, when related to hardware the budget is a provided budget. Four basic budget types are distinguished and referred to by the names *cpu*, *mem*, *stor* and *net*. The values of these budgets either have a fixed value, or they depend on the values of other budgets. If the budget of a parent model element (with no children) has not been specified, it gets assigned the sum of the values of its children.

The allocation specification relates software model elements to hardware model elements. Furthermore, it defines which budgets may take part in a constraint. Model elements can take part in an allocation relationship as a whole, but parts of model elements can also be allocated. The ratios defining the parts, as well as the number of children of the hardware model element, define the allocation factor. When including the software element budget in a constraint, its value is multiplied by the allocation factor.

Constraints define relations between required and provided budgets. They provide a means to impose restrictions on the system. A constraint is specified as a boolean expression which can contain references to budgets.

# Chapter 6

# System Model Viewpoint

This chapter describes the system model viewpoint, which corresponds to the third sub-problem of the problem statement of Section 1.2. We introduce a UML extension in order to automatically recognize the structure of a system, calculate budget values and evaluate constraints. This chapter thus contains a UML implementation of the concepts presented in Chapter 5.

## 6.1 Introduction

Besides textual descriptions of systems, models are becoming increasingly important as design artifacts in the Model Driven Architecture (MDA) [17]. For software systems, the UML is widely used as a modeling language. Since a CMS is not purely a software system, SysML would be the modeling language of choice. Recall that a lack of tool support for SysML (Section 3.4) is the reason that models are expressed in UML. To show the close relationship between UML and SysML, the SysML modeling constructs are provided with the UML modeling constructs in this chapter.

Instead of expressing performance properties of a CMS in an Excel sheet, which is large, complex and difficult to understand, a UML model provides a much clearer overview of the structure of a system. Allocation of software to hardware is clearly visible, because of the allocation relationships. The use of UML models to express the structure and performance properties of systems results in a better understanding of those systems, making it easier to experiment with different structures, budgets and constraints.

Even though Rational Software Architect does provide good UML modeling support for a variety of models and model elements, support for UML objects is poor. In RSA, an object, an instance of a class, does not have access to the attributes defined in the class. Hence, different objects of a single class cannot assign different values to an attribute defined in the class.

Since no UML objects can be used, all instances are modeled as UML *classes*. To still distinguish different model elements classes from each other, stereotypes have been defined for the concepts of Chapter 5, which are described in the following sections. All stereotypes provide extensions to the UML metaclass Class, as described in Section 3.1.1. Whereas one would like to describe real-world objects as UML objects, they are described using UML classes, thereby essentially shifting the levels of the four-layer metamodel hierarchy one level up. The instances of all model elements are defined at level M1, leaving level M0 empty.

## 6.2 TProfile Definition

To distinguish the model elements created in this chapter from standard UML elements, stereotypes are defined which correspond to the elements of the domain model. To group all stereotypes presented in this chapter, the Thales profile, or *TProfile* has been created. A profile is essentially a UML package, only stereotyped as «profile».

The complete TProfile is shown in Figure 6.1. It contains the stereotypes corresponding to the domain model elements for both hardware («THWSystem» and «TNode»)and software («TSWSystem», «TLayer» and «TComponent»). They are all subclasses of the abstract stereotype «*TBudgetElement*», which extends the UML Class. This means that all subclasses of the «*TBudgetElement*» can be assigned to UML classes.

The «allocate» stereotype extends the UML Association. Associations in a model can thus be assigned this stereotype when the TProfile is applied.

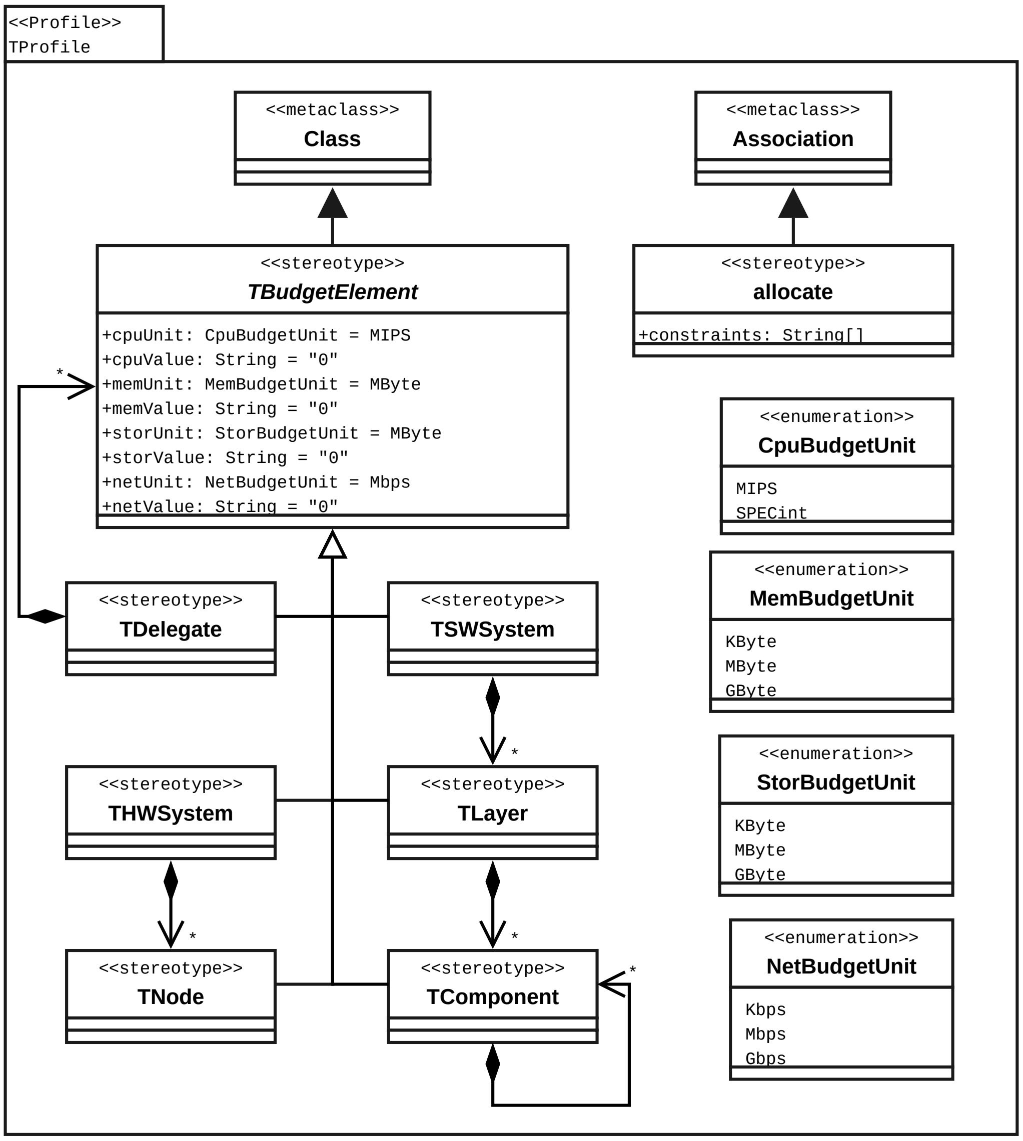


**Figure 6.1** — The complete TProfile

The following sections discuss the various parts of the TProfile in detail.

## 6.3 Hardware

This section describes the hardware part of a system, which consists of a number of nodes connected by a network.

### 6.3.1 Mapping Domain Concepts into UML Equivalents

The two main domain concepts of the hardware are the *hardware system* itself and the *nodes* of which it is composed. Both are represented by UML classes and are related via a composition relationship.

#### Hardware System

A hardware system is the high-level entity that groups a number of nodes. With SysML, it would be modeled by a block. Since the SysML block is basically a UML class, the hardware system is modeled by a class. The class modeling the hardware system is a level M1 model element, as shown in Figure 6.2.

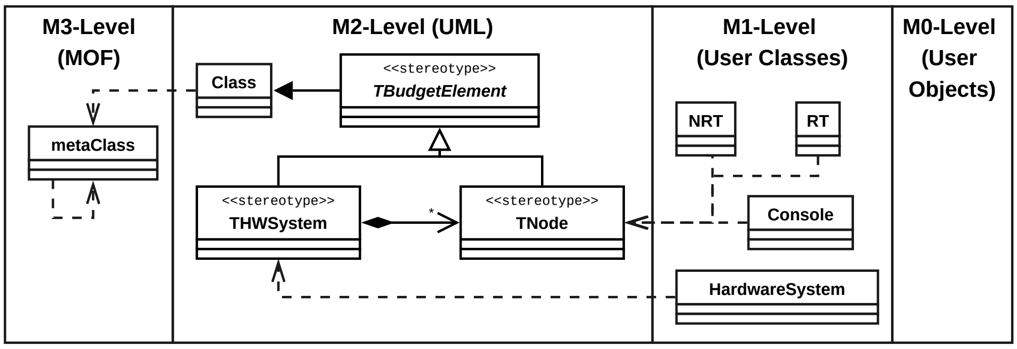


**Figure 6.2 —** The hardware model elements placed in the OMG four-layer metamodel hierarchy. The instances (e.g. HardwareSystem) resides at level M1 and the stereotypes, which extend the UML Class, at level M2. Level M0, the user level, remains empty.

As shown in Figure 6.2, the stereotype for the hardware system is called «THWSystem».

**«THWSystem»** The hardware system is represented by a UML class. To identify the class as representing a hardware system, it is stereotyped as a «THWSystem».

| Stereotype | Base Class | Tags |
|---|---|---|
| «THWSystem» | Class | |

**Table 6.1 —** «THWSystem» stereotype definition

#### Node

Using SysML, any part of the hardware would be represented by a block. Nodes are part of the hardware system and would be modeled by blocks as well. Similarly to the hardware system, nodes can be represented by classes. Because the hardware system itself is also represented by a class, the composition relationship between the two can easily be made.

As shown in Figure 6.2, the class modeling a node, for instance a Console, is a level M1 element as well. For each type of node, a class is created at level M1. For the example CMS, three nodes would be created: Console, RT and NRT. The stereotype representing a node is called «TNode».

**«TNode»** Nodes from the domain model are represented by UML classes. To distinguish them from standard UML classes, they are stereotyped as «TNode».

| Stereotype | Base Class | Tags |
|---|---|---|
| «TNode» | Class | |

**Table 6.2 —** «TNode» stereotype definition

### 6.3.2 Example CMS Hardware Structure Model

An example application of the stereotypes defined for the hardware system is shown in Figure 6.3. Three instances of the «TNode» stereotype are shown in the figure, one for each type of node in the example CMS. Of the Console node type, there are eight nodes in the example CMS. This is reflected in the model by the multiplicity of 8 at the composition relationship to the Console model element.

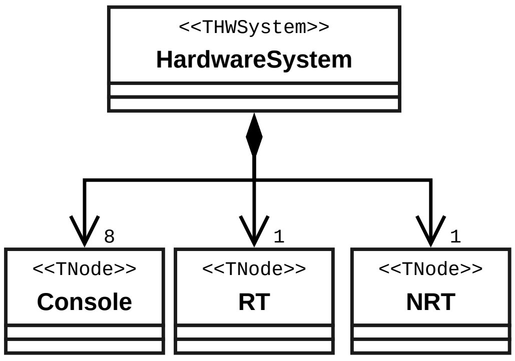


**Figure 6.3 —** The structural UML model of the hardware of the example CMS.

## 6.4 Software

As discussed in Chapter 5, the main concepts of the software are *components*, *layers* grouping those components and the *software system* itself, grouping layers.

### 6.4.1 Mapping Domain Concepts into UML Equivalents

The sections below discuss the domain concepts of the software model and provide corresponding SysML and UML elements.

### Software System

The properties of the software system are similar to those of the hardware system presented in Section 6.3. Whereas the hardware system groups nodes, the software system groups all layers in the system.

Using SysML, a software system would therefore be modeled by a single block, which would contain the model elements representing the layers. In UML however, the software system is represented by a class, just like the hardware system.

The position of the software system in the OMG four-layer metamodel hierarchy is similar to the hardware system, as illustrated in Figure 6.4. It does also not have a real-world instance and exists only at level M1.

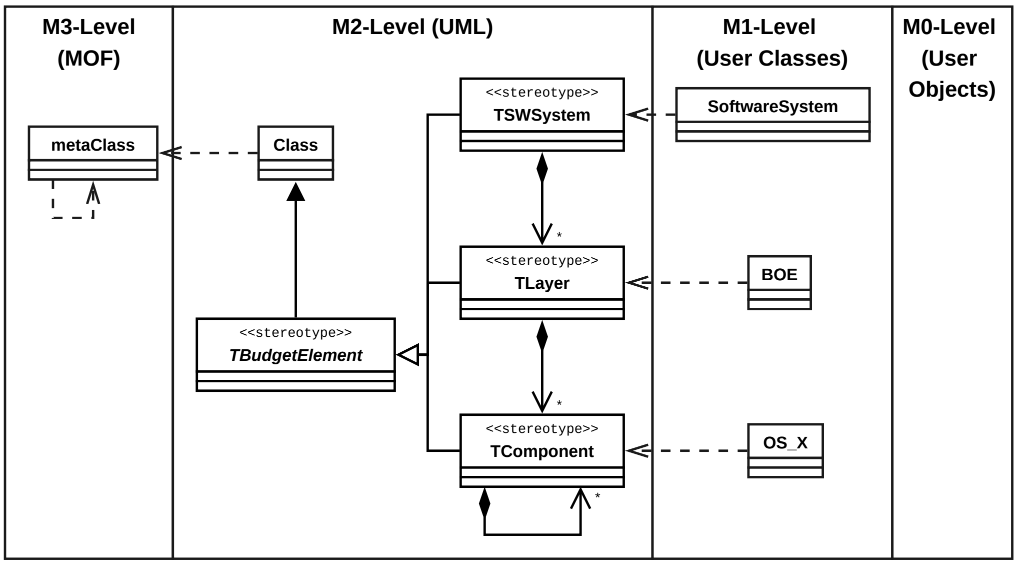


**Figure 6.4 —** Place of the software part of a system in the OMG metamodel hierarchy (Dashed lines show instance-of relationships)

**TSWSystem** The software system is represented by the «TSWSystem» stereotype. The stereotype itself does not have any tags and is only included for the identification of a software system. To ensure that the «TSWSystem» stereotype can be applied to a class, it extends the abstract «*TBudgetElement*» stereotype.

| Stereotype | Base Class | Tags |
|---|---|---|
| «TSWSystem» | Class | |

**Table 6.3 —** «TSWSystem» stereotype definition

### Layer

Abstract concepts such as layers can be represented in SysML by blocks, because blocks are meant to represent both concepts and real-world entities. Layers group components in different levels of abstraction and are therefore important to structure a model. Furthermore, layers have to contain budget information. Either for a layer itself (for instance when no components have been defined within the layer), or as a summary of the budget information of its contained components.

The way layers are included in the model of this section provides many degrees of freedom to the engineer, because they have many of the properties of a component. Instead of introducing yet another model element with new properties, model element properties are consistent throughout the system, thus easing the development of systems using the model.

**TLayer** A layer is represented by a UML class as well, as illustrated in Figure 6.4. To denote that a class represents a layer, it is stereotyped as a «TLayer». The stereotype extends the abstract «*TBudgetElement*» stereotype and does not have any tags.

| Stereotype | Base Class | Tags |
|---|---|---|
| «TLayer» | Class | |

**Table 6.4 —** «TLayer» stereotype definition

#### Component

A component is a modular part of a system, which provides some service to the system and can consist of several other parts (which are components themselves in turn). Just as with the other model elements, the SysML model element suitable to represent such a component is the *Block*.

The UML equivalent of a SysML block is a class, as blocks are in fact classes which have the «Block» stereotype applied. For this reason, components are represented using UML classes, which provide all required features. Composition of components can be accomplished by the use of the composite pattern, which is applicable to blocks as well as classes.

The representation of components with classes is illustrated in Figure 6.4. All instances of the components are placed in the OMG four-layer metamodel hierarchy at level M1. The exact number of instances of each component is not known until the allocation of the components to nodes has been specified. This is further discussed in Section 6.6.

**TComponent** UML Classes that represent Components are stereotyped as «TComponent». The stereotype extends the «*TBudgetElement*» stereotype and does not have any tags.

| Stereotype | Base Class | Tags |
|---|---|---|
| «TComponent» | Class | |

**Table 6.5 —** «TComponent» stereotype definition

### 6.4.2 Example CMS Software Structure Model

An example application of the stereotypes for the software system is shown in Figure 6.5. The software system consists of three layers: CSCIs, Middleware and BOE. The layers consist of a number of components, however, for illustration purposes only three components are shown for the layers CSCIs and BOE.

The composition relationships between the model elements all have a standard multiplicity of 1. However, just as with the hardware, other values can be specified to introduce information about the number of instances of a particular model element. For software however, this will generally never be the case and the exact semantics of such a multiplicity are not defined for software either. Still, including a multiplicity value other than 1 does have some effect on the calculation of budgets, as explained in Section 6.5.

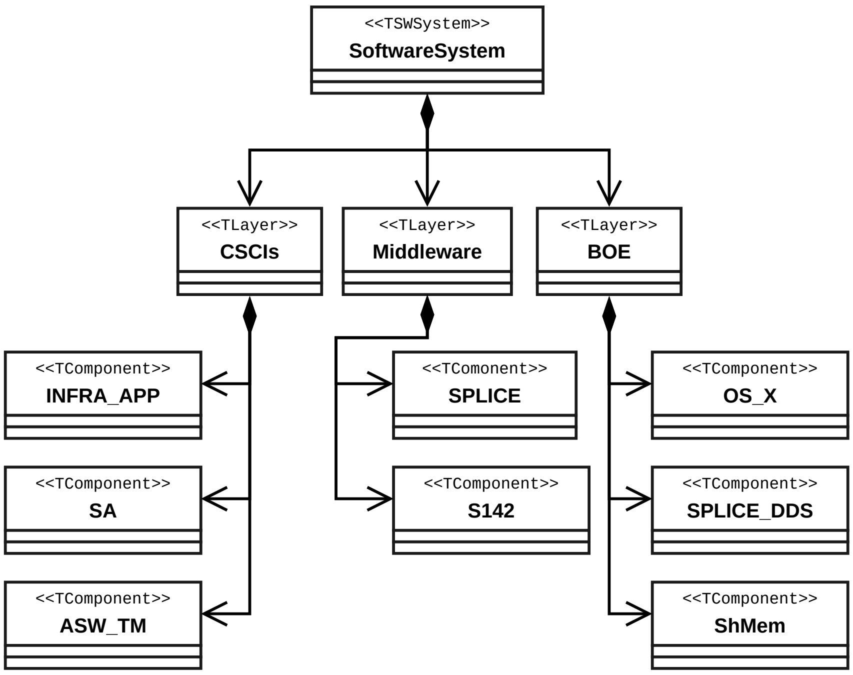


**Figure 6.5 —** The structural UML model of the software of the example CMS.

## 6.5 Budgets

Resources are quantified by budgets, which all have types, units and values. Budgets are present in all parts of the model of a system, as for both hardware and software budgets need to be defined. The difference between required and provided budgets is not made explicitly when designing a system model. Instead, a budget is presumed to be a required budget when it is part of the software system. Vice versa, a budget is presumed to be a provided budget when it is part of the hardware system. The distinction between hardware and software systems is made clear though through application of stereotypes, as described in Sections 6.3 and 6.4.

### 6.5.1 Mapping Domain Concepts into UML Equivalents

Budget information needs to be assigned to a variety of model elements. As described in Sections 6.3 and 6.4, model elements are represented by stereotyped UML classes.

In SysML, the model elements would be represented by blocks, which have a special predefined «values» compartment. A value is a special kind of attribute of the block, which can be specified in more detail using new blocks. The best way to represent budget information for a block would be to put it in the «values» compartment.

Values of SysML blocks are essentially UML attributes, so budget information is added to UML classes by means of attributes. To ease the creation of a model of a system, four default attributes are available for each model element, representing the four basic budgets for cpu, memory, storage and networking. To make these budgets available to all model elements by default, they have been added as tags to the *«TBudgetElement»* stereotype, the supertype for all other model elements.

The overview of the TProfile in Figure 6.1 shows the tags of the four budget types. One tag for the unit of each budget and one tag for each value. The available units for each budget type are defined by the enumerations. For instance, for the budget of the CPU this means that there are two tags: *cpuUnit* and *cpuValue*. The complete description of the stereotype is listed in Table 6.6. For each budget type, an enumeration is defined listing

| Stereotype | Base Class | Tags |
|---|---|---|
| «*TBudgetElement*» | Class | cpuUnit<br>cpuValue<br>memUnit<br>memValue<br>storUnit<br>storValue<br>netUnit<br>netValue |

**Table 6.6 —** «*TBudgetElement*» stereotype definition

the available units. They are listed in Table 6.7.

| Enumeration | Literals |
|---|---|
| CpuBudgetUnit | MIPS, SPECint |
| MemBudgetUnit | KByte, MByte, GByte |
| StorBudgetUnit | KByte, MByte, GByte |
| NetBudgetUnit | Kbps, Mbps, Gbps |

**Table 6.7 —** TProfile unit enumerations

Four of the eight tags added to the «*TBudgetElement*» stereotype are for the definition of the values, four are for the corresponding units. Tags defining values are of the *String* type, the tags defining the units are typed by the previously defined enumerations. The specification of the tags is shown in Table 6.8.

| Tag | Type | Multiplicity | Default value |
|---|---|---|---|
| cpuUnit | CpuBudgetUnit | [0..1] | CpuBudgetUnit::MIPS |
| cpuValue | String | [0..1] | "0" |
| memUnit | MemBudgetUnit | [0..1] | MemBudgetUnit::MByte |
| memValue | String | [0..1] | "0" |
| storUnit | StorBudgetUnit | [0..1] | StorBudgetUnit::MByte |
| storValue | String | [0..1] | "0" |
| netUnit | NetBudgetUnit | [0..1] | NetBudgetUnit::Mbps |
| netValue | String | [0..1] | "0" |

**Table 6.8 —** «*TBudgetElement*» tag definitions

### 6.5.2 References

No two names of budgets of a single model element are allowed be equal. This is already forced by the fact that budgets are modeled as attributes, which are not allowed to have equal names within a class as well. Furthermore, a name can consist of letters (lowercase and capital), digits and underscores, but always has to start with a letter. The uniqueness of a budget name within a model element allows for reference by name.

Referencing a budget from within the model element it contains, can simply be done by referring to the name of the budget. Consider the memory budget of the INFRA_APP component. From within this component, it can be referred to by the name *mem*. To explicitly refer to the budget with the specified name *within* the same model element, the budget name can also be prefixed with *this*. A reference to the memory budget of the same model element can thus also be specified as *this.mem*.

When including a reference to the same budget from another component, the budget name has to be prefixed with the name of the component. Components of a system are all in the same namespace and thus need to have unique names throughout the system. A reference to the memory budget of the INFRA_APP component from another component is *INFRA_APP.mem*.

### 6.5.3 Values

As discussed in Section 5.4.2, budgets have either fixed values, or their values depend on other budgets. In both cases, the value and unit of a budget are specified using the tags of the applied stereotype. The tags representing the budget values have the *String* type, which means that a reference to some other budget can also be a budget value.

#### Fixed

In case of a fixed value, it is directly specified in the appropriate tag of the corresponding stereotype. For example, to set a memory budget of 100 MByte for the SA component, the value of the tag *SA.memValue* is set to 100 and the value of the tag *SA.memUnit* is set to *MemBudgetUnit::MByte*.

#### Dependent

Dependent budget values are budgets that reference other budgets in their definitions. Referred budgets are either contained by the same model element as the calling budget (a local reference), or they exist in other model elements (remote reference). Local references can always be made, as long as the referred budget exists. Remote references require the existence of a *dependency* relationship to the model element containing the remote reference. The remote budget can be referred to by prefixing its name with the name of the owning model element.

A shorter reference method is available, which does not require the name of a remote budget to be prefixed with the name of the model element. It can however only be used if the source model element does not contain a budget with the same name as the referred budget. In Figure 6.6 for example, a reference from INFRA_APP to S142.avg_overhead can be made by simply stating *avg_overhead*. This is only allowed if INFRA_APP itself does not contain a budget with the name avg_overhead and if no other dependency of INFRA_APP contains a budget with that name.

An example of dependency relationships is illustrated in Figure 6.6. It shows dependency relationships between the components of the CSCIs layer and the S142 component of the middleware.

Since budget values have the *String* type, the basic arithmetic operations of addition, subtraction, multiplication and division can be used to combine and change budget values. For a complete and exact specification of the syntax, please refer to Appendix A.

#### Sum

When no value has been defined for a budget by the engineer, it is derived from the budget values of the children of the owning model element. If a value has been defined, but the sum of the budget values of the children is greater, the defined value is overruled by the summation. The value of the budget thus is the maximum of its own value and the sum of its children.

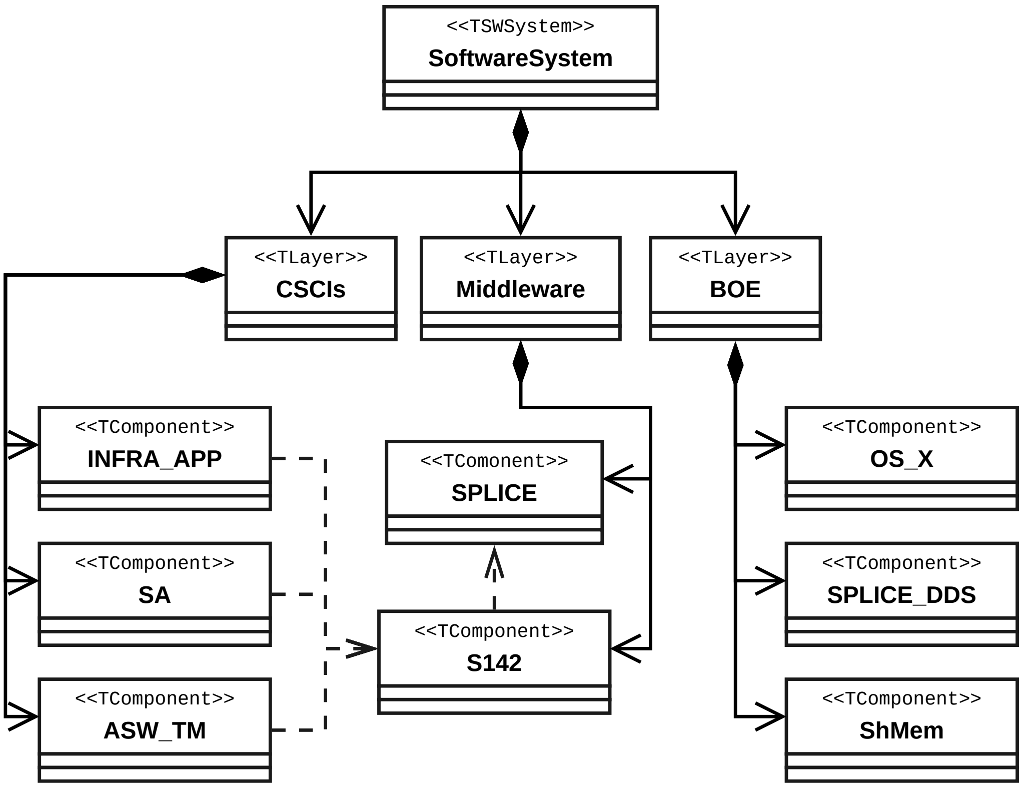
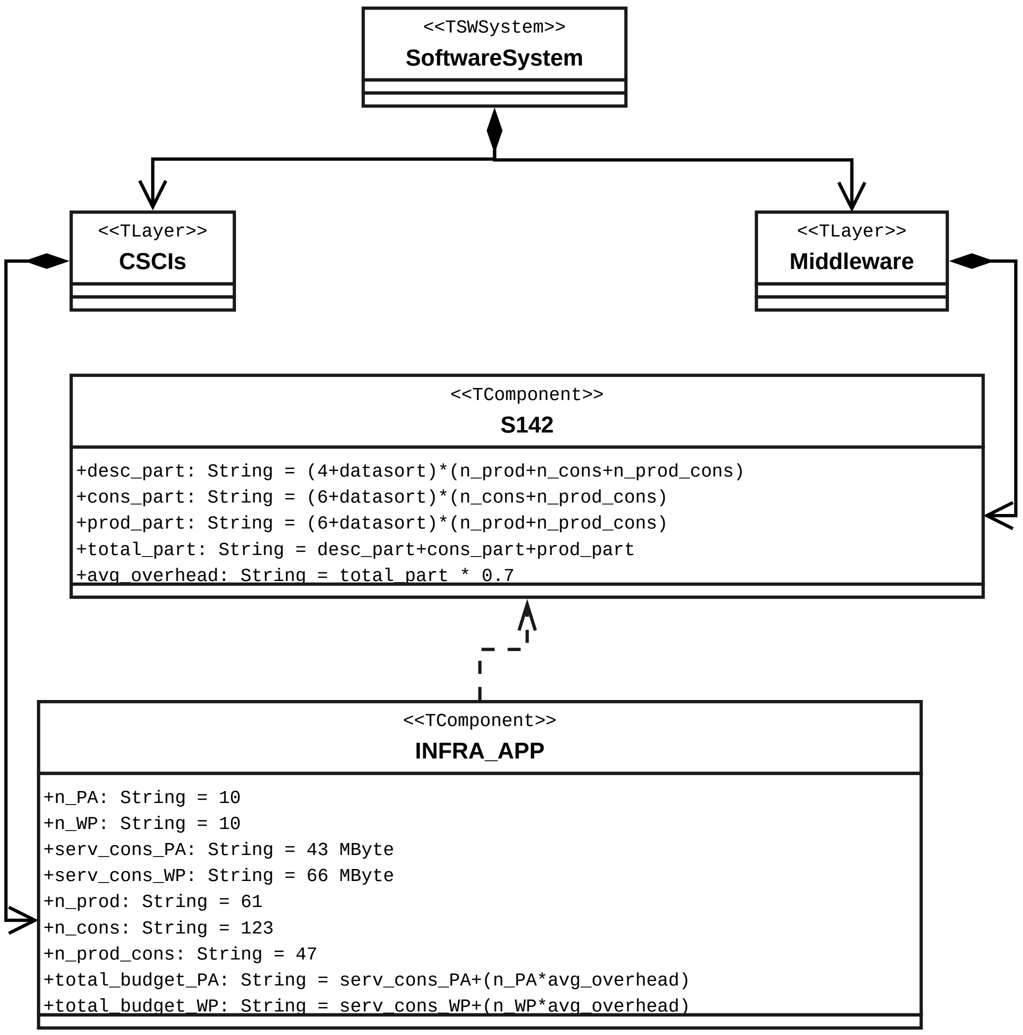


**Figure 6.6** — Example dependency relationships between the CSCIs and S142 and between S142 and SPLICE.



<<TSWSystem>>
SoftwareSystem
<<TLayer>>
CSCIs
<<TLayer>>
Middleware
<<TComponent>>
S142
+desc_part: String = (4+datasort)*(n_prod+n_cons+n_prod_cons)
+cons_part: String = (6+datasort)*(n_cons+n_prod_cons)
+prod_part: String = (6+datasort)*(n_prod+n_prod_cons)
+total_part: String = desc_part+cons_part+prod_part
+avg_overhead: String = total_part * 0.7
<<TComponent>>
INFRA_APP
+n_PA: String = 10
+n_WP: String = 10
+serv_cons_PA: String = 43 MByte
+serv_cons_WP: String = 66 MByte
+n_prod: String = 61
+n_cons: String = 123
+n_prod_cons: String = 47
+total_budget_PA: String = serv_cons_PA+(n_PA*avg_overhead)
+total_budget_WP: String = serv_cons_WP+(n_WP*avg_overhead)

**Figure 6.7 —** CMS budget calculation of INFRA_APP component. This part of the complete CMS model shows fixed budgets (INFRA_APP.n_PA) and dependent budgets (INFRA_APP.total_budget_PA).

### 6.5.4 Extra Budgets

Besides the budgets that are predefined in the stereotypes, the engineer has the possibility to introduce extra budgets and parameters. Parameters are also called *budgets*, which means that all values are specified using budgets and calculations are only made with budgets.

In contrast to the predefined budgets, these extra budgets can have, but do not need to have a type or unit. This means that simple amounts (which do not have a type or unit) can be specified as well. When a budget without type or unit is combined with a budget with a type or unit (for example in a multiplication), the resulting budget will always have a type or unit. Multiplying a memory budget of 100 MByte with a parameter 5 will thus result in the memory budget of 500 MByte.

New budgets can be added to a model element by adding an attribute to the class representing that model element. This is possible for all model elements of hardware and software.

### 6.5.5 Example CMS Budgets

To illustrate how budgets are specified for the example CMS, the memory budget calculation of the INFRA_APP component is shown in Figure 6.7.

The example shows fixed budgets (for instance *INFRA_APP.n_PA* and *INFRA_APP.serv_cons_PA*) and budgets that depend on other budgets within the same model element (for instance *S142.total*). It also shows budgets that depend on other budgets of elements connected by a dependency relationship (for instance *INFRA_APP.total_budget_PA*).

## 6.6 Allocation

Allocation is the way to relate a software system to a hardware system. This can be modeled in several ways. SysML, for instance, provides the possibilities to use a table, matrix, allocate relationship or even the definition of an allocation in the special predefined block compartments «allocatedFrom» and «allocatedTo».

Software elements can be allocated to hardware either as a whole or in parts, which corresponds to simple and complex allocation respectively. Many parts of the model of this chapter are visually, rather than textually, presented to the engineer. This provides him with an easily surveyable model of the system. To keep as most parts of the model as possible represented visually, allocations are also not expressed using text or tables.

### 6.6.1 Mapping domain concepts into UML equivalents

In UML the allocation specification remains close to its SysML equivalent of an allocation relationship. Allocating a software model element to a hardware model element is accomplished by an association between the two. Since allocation always defines that software is allocated to hardware and not the reverse, the association is not required to be directed. However, to distinguish the allocation association relationship from a regular association, it is stereotyped.

The «allocate» stereotype defines that an association between two model elements is the allocation of one element to the other. Allocation relationships always have to relate exactly one software element to exactly one hardware element.

| Stereotype | Base Class | Tags |
|---|---|---|
| «allocate» | Association | |

**Table 6.9 —** «allocate» stereotype definition

The «allocate» stereotype is included in the TProfile. It provides an extension to the Association metaclass, as shown in Figure 6.1.

### 6.6.2 Example CMS Allocations

An example of an allocation is depicted in Figure 6.8. It shows the allocation of the CSCIs layer to the complete hardware system. This is just for illustration purposes, because in the examle CMS, the CSCIs layer does not need to be allocated to the NRT server. The proper allocation of the CSCIs layer requires a delegate and is described in Section 6.7.1.

Three allocation relationships are included and together define the allocation of the CSCIs layer. The sum of the ratios at the CSCIs equals 1.0, denoting that the layer is completely allocated. However, not all components of the CSCIs are equally allocated, hence the three allocation relationships.

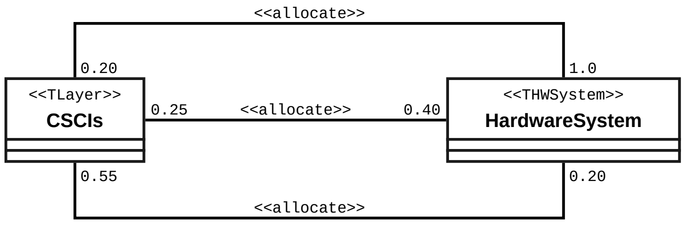


**Figure 6.8 —** Example allocation in the example CMS of the CSCIs layer to the hardware system.

## 6.7 Delegates

The hardware and software structures described in Sections 6.3 and 6.4 define a strict hierarchy in the hardware and software systems. Grouping concepts such as the hardware system, software system or layers will always include all of their contained nodes, layers or components. To provide the engineer with a way to group an arbitrary number of model elements, the *delegate* model element is available.

Although the delegate can contain all types of model elements, it should *not* mix up elements of hardware and software. Delegates can however contain other delegates, as long as hardware and software are kept separated.

The delegate is a special model element and has special properties. There are similarities between a delegate and 'regular' model element:

- A delegate can contain other model elements, just like the hardware system, software system, layer and component model elements.
- A delegate owns the same basic budget types as other model elements. Cpu, mem, stor and net are always available. Custom budgets can be added to a delegate too.

Besides these similarities, there are some important differences between delegates and other model elements:

- A delegate is always part of either the software system or the hardware system, no matter what elements it contains. If the delegate contains software elements, it is part of the software system. If it contains hardware elements, it is part of the hardware system.

To include delegates in the model, the TProfile is extended with the additional stereotype «TDelegate».

| Stereotype | Base Class | Tags |
|---|---|---|
| «TDelegate» | Class | |

**Table 6.10 —** «TDelegate» stereotype definition

An extension is made to the TProfile, thereby including the new «TDelegate» stereotype. It has to have the basic budget types and therefore extends the «*TBudgetElement*» stereotype, as illustrated in Figure 6.1.

Delegates can contain all other model elements, including other delegates, but a delegate is *not* allowed to contain itself.

### 6.7.1 Example CMS Delegates

Instead of allocating the CSCIs layer to the complete hardware system (as described in Section 6.6.2), the layer has to be allocated to the Console nodes and the RT node exclusively. To model this, a delegate is required which groups the Console nodes and the RT node, but excludes the NRT node. This is illustrated in Figure 6.9. Instead of defining allocation relationships between the CSCIs and the hardware system, they can be defined between the CSCIs and the new Console_RT delegate, thus allocating the CSCIs to the Console nodes and the RT node only.

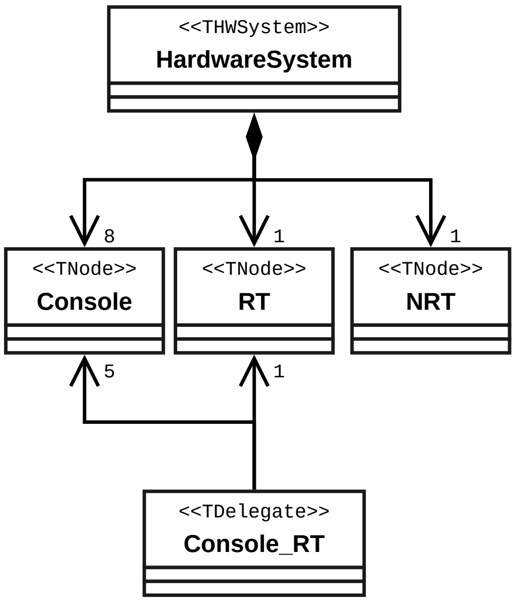


**Figure 6.9 —** A delegate in the example CMS

## 6.8 Packages

For a better overview of large and complex models, model elements can be grouped using packages. This is especially useful when creating models in RSA, as described in Chapter B. Large models tend to become complex and hard to survey. Grouping model elements in packages can reduce cluttering, which improves readability of the model.

## 6.9 Constraints

Constraints relate required budgets to provided budgets. In SysML, budgets would be defined in the «values» compartment of a block. Constraints on these budgets are defined using constraint blocks, which contain both the constraint expression and a list of parameters. The model element representing the application of a constraint block is called a *constraint property* and can be included in a parametric diagram. In a parametric diagram, the budget values (represented by blocks containing the fully qualified name of the budget, recall Figure 3.11) are bound to constraint parameters via the ports owned by the constraint properties.

### 6.9.1 Mapping domain concepts into UML equivalents

Since the UML does neither contain a parametric diagram nor constraint blocks, this method cannot be applied. Instead, the constraints can be defined in two rather different ways: by means of a UML activity diagram, or by an extension of the TProfile.

#### UML Activity Diagram Approach

The functionality provided by a parametric diagram can be imitated by a UML activity diagram. The Call Behavior Action can be considered the UML equivalent of a SysML constraint block. Input pins can be used to model the parameters of a SysML constraint block and can be bound to budget values.

Whereas in a parametric diagram, budgets are represented by a block with the fully qualified name, in UML they can be defined by data stores. By using the fully qualified name of the budget for the data store, it can represent the corresponding budget. The data stores can be connected to the input pins of the activities by object flows, thus realizing the binding of budget values to the parameters of the constraints.

To distinguish an action representing a constraint from a regular action, it is stereotyped as «TConstraint». The body of constraint is specified in an attached UML constraint (essentially a note), as illustrated in Figure 6.10.

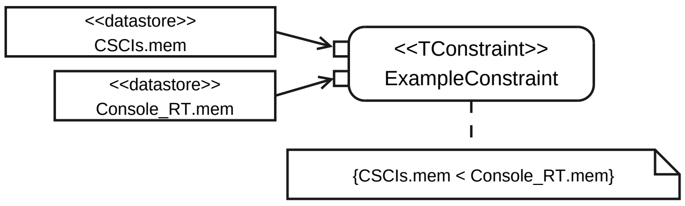


**Figure 6.10 —** Constraint definition using the UML activity diagram

#### Profile Extension Approach

Constraints always define relations between required and provided budgets and do therefore reside on the boundary between hardware and software. In fact, only budgets of model elements that take part in an allocation relationship can be constrained. Constraints can thus also be specified on the allocation relationships themselves, for instance by means of extending the «allocate» stereotype with an additional tag.

When multiple allocation relationships are defined between two model elements, constraints can be added to any of those relationships. There is no difference between adding a constraint to one allocation relationship or adding it to another, as long as the relationships relate the same two model elements.

Even though the constructs provided by the parametric diagram can be mimicked by the UML activity diagram, this approach introduces a lot of overhead in defining constraints and is therefore not preferred. To create the diagram of Figure 6.10, as many as nine elements have to be created by the engineer (this includes the data stores, constraint, call behavior action, two input ports and all relationships). An extension of the «allocate» stereotype, however, only introduces a minimal overhead with respect to the current model. Section 6.9.2 describes this approach.

### 6.9.2 TProfile Extension

In order to add constraints to allocation relationships, the «allocate» stereotype (added to the TProfile profile in Section 6.6) is extended with the *constraints* tag. This is specified in Tables 6.11 and 6.12.

| Stereotype | Base Class | Tags |
|---|---|---|
| «allocate» | Association | constraints |

**Table 6.11 —** «allocate» stereotype definition, including constraints

| Tag | Type | Multiplicity |
|---|---|---|
| constraints | String | [0..*] |

**Table 6.12 —** «allocate» constraint tag definition

The constraints tag of the «allocate» stereotype is shown in the TProfile of Figure 6.1. If multiple «allocate» relationships exist between a software element and a hardware element, the constraints can be placed on any of them. When evaluating the constraints, they are all processed, no matter on which «allocate» relationship they reside.

### 6.9.3 Constraint Syntax

Just as budget values, constraints are represented by expressions of the *String* type. The syntax of these expressions is similar to the syntax of the expressions for budget values. The main difference is that a constraint expression always has to have a boolean result, so it has to yield either *true* or *false*. For the exact syntax of constraint expressions, please refer to Appendix A.

## 6.10 Discussion

The elements introduced in this chapter mostly follow the concepts of the domain viewpoint. However, at some points the system model differs from the domain viewpoint. The most important decisions which have led to the current model are discussed in this section.

### 6.10.1 Layers

It is easy to see how a software system can be decomposed into a number of components. Layers are, however, also introduced as model elements and reside in between the software system element and component elements. In fact, a layer does not add to the functionality of the system, but only exists as a grouping concept for components providing functionality at the same level of abstraction. Despite a layer having only the *level* attribute, it is still included in the model.

Engineers do have the concept of a layer in mind when designing a system. It is a powerful tool to ease system design and therefore has to be included in the model. The level attribute of the layer however, is not included. Levels can be used to define an ordering of the layers and thereby, implicitly, define restrictions on the communication between components in those layers. The model described in this chapter has to allow engineers to experiment with the structure of a system and provide as much freedom in doing this as possible, while still providing the engineer with a clear picture of the structure. Addition of a *level* attribute to a layer does not introduce new possibilities to the engineer, but would only restrict the freedom of experimentation. Dependency relationships can be used to create similar constraints on the model, but have far more potential.

The addition of budget information to a layer *does* improve the model, because it enables the engineer to treat a layer similar to any other component and design a system at a very high level (i.e. the level where a single layer can exist without any contained components).

### 6.10.2 Delegates

A delegate is not an element of the domain viewpoint, but is important in the system model. When allocating software to hardware, the domain model simply states that it is possible to allocate every software element or collection of elements to any hardware element or collection of elements. To implement this, the delegate is required as an additional model element in the system model viewpoint.

The delegate is required when an allocation is to be defined of other than default collections of elements. A layer can for example be looked upon as a default collection of all components it contains. Also, the hardware system is the default collection of all nodes it contains.

To allocate some software components to a subset of all nodes, the structure of the hardware model is not to be changed. Instead, a delegate element is used to delegate the allocation of the software components to the designated hardware nodes. Since, in contrast to nodes being parts of the hardware system, they are not part of the delegate element, relationships between the delegate and elements are directed associations instead of composition relationships.

### 6.10.3 Objects

All parts of the model presented in this chapter are defined at level M1 of the OMG meta-model hierarchy, whereas one would expect to find elements (objects) at level M0 as well. After all, the model has to be a reflection of the real world and should define instances of the classes of level M1. The main reason for this is, again, the RSA tool, which has no proper implementation of UML objects. Class Instances, the UML objects in RSA, have no connection with the classes they instantiate. Attributes of a class are *not* known by the instance of that class, which means that no specific values can be set for those attributes either. Including class instances in the model does not contribute to the ease of use of the model, because of this deficiency in RSA. They are therefore not included.

Multiplicities at composition relationships provide a means to fill this void. The interpretation of a multiplicity of $x$ at a composition relationship is that for every instance of the parent, $x$ instances of the child exist in reality (at level M0). Since a hardware system is usually only instantiated once, the multiplicities at the nodes do define the number of instances of those nodes in the real world.

Multiplicities can also be specified in the structural model of the software, where they have the same consequences as in the hardware model. Even more important than with the hardware system is the question whether a software model element will ever be instantiated more than once with respect to its parent. The option remains available due to the similar implementation of the hardware model and the software model, but is just not likely to ever be used.

### 6.10.4 Constraints

The implementation of most domain concepts in UML is almost identical to the SysML implementation. Especially the implementation of the structure closely resembles the SysML approach, since SysML blocks are essentially UML classes. In SysML, constraints would be implemented using constraint blocks and parametric diagrams, which are both new concepts of SysML.

The UML diagram which resembles the SysML constraint blocks and parametric diagrams closest is the activity diagram. With it, the same modeling constructs can be accomplished, namely binding attribute values to formal parameters of constraints. A drawback of this approach is that it requires a large amount of UML elements for each constraint. Whereas in SysML a constraint can be modeled by a single constraint block, the activity diagram approach in UML would require a CallBehaviorAction *and* an input pin for each parameter of the constraint. Furthermore, the expression representing the constraint cannot be included in the CallBehaviorAction, but would have to be specified in a separate UML constraint.

The close resemblance of the activity diagram approach to the use of SysML constraint blocks does not make up for the complexity of creating such a model. It is for this reason that constraints are part of allocation relationships.

# Chapter 7

# Tool Support

This chapter describes the requirements of the System Verifier Tool (referred to as *the tool* in this chapter) supporting the creation of the UML model described in Chapter 6. Subsequently, the implementation of the tool itself is discussed in detail. With our own expression-based language, created using the parser generator ANTLR, budget values and constraints can easily be expressed and evaluated.

## 7.1 Rational Software Architect

The Integrated Development Environment (IDE) Rational Software Architect is built by IBM as a set of plugins for the Eclipse platform and provides the same basic functionality as Eclipse. RSA adds a number of extra features, of which the most important one consists of the visual creation of UML models. UML 2.0 is largely supported, including the profiling extension mechanism.

### 7.1.1 UML Modeling

RSA applies the Model-View-Controller (MVC, [4]) pattern in its UML modeling functionality. The distinction between the model and the views on this model is especially important when creating a UML model.

The model part of the MVC implementation is realized by the UML2 package [31] and provides a representation of a UML model in Java classes. This is available to all versions of Eclipse, also to the non-IBM implementations. RSA provides a view and controller for UML models, which, among other things, distinguishes it from the standard version of Eclipse.

## 7.2 Tool Purpose

The main purpose of the tool is to support systems engineers in the initial design of a system, consisting of software and hardware. To get an early indication of whether the hardware can provide enough resources to the software, the tool helps the engineers with the following four tasks:

1. Visual creation of a UML model of a system, consisting of both hardware and software and of allocations.

2. Specification of required and provided budgets, budget values and budget dependencies.

3. Specification of constraints on the budgets.

4. Automatic verification of the specified constraints.

An implementation of the model by the tool ensures that these tasks can be performed. The model of a complete system consists of a hardware model and a software model, connected by allocation relationships. This is necessary to complete task 1. These models are enhanced with budget information (task 2) and constraints are defined on the budgets (task 3). Finally, the tool provides for an automatic verification mechanism to verify the constraints (task 4).

The basic functionality required for completing the first task is already provided by RSA, because it supports the visual creation of UML models. The additional value of the tool described in this chapter is that it makes the TProfile available, which allows the engineers to describe systems according to the model described in the previous chapters. The profile is also required for tasks two and three. The functionality to complete the fourth task is completely provided by the tool.

## 7.3 Requirements

For the tool to meet its stated purpose, several requirements have to be fulfilled. The first, important requirement is the following:

**Requirement 1** *The tool has to integrate with RSA.*

For the creation of UML models, engineers use Rational Software Architect. Because the engineers are already used to working with RSA, extension of the IDE reduces the barrier to start using the tool.

Since RSA is Eclipse-based, it can easily be extended by means of plugins [3]. A big advantage of developing the tool as an RSA plugin is that it can easily be integrated with existing installations, so no additional software needs to be installed.

**Requirement 2** *The tool has to provide support for visual design of a structural model of a system.*

Eclipse plugins have access to almost all parts of the Eclipse framework and other installed plugins. The visual UML modeling functionality, as well as the underlying UML 2.0 implementation [31], integrated into RSA can be accessed as well by custom plugins.

To support UML extensions, RSA provides the option to create custom UML profiles. These profiles can be applied to UML models, so the stereotypes defined in a profile can be applied to the elements of the model. Profiles can be bundled with a plugin, thus simplifying distribution of the profile. The model created in Chapter 6 is a UML model with an applied profile and can thus be created in RSA.

The combination of basic RSA UML modeling functionality and the introduction of a UML profile is sufficient to support visual design of a structural model of a system.

**Requirement 3** *Created UML system models have to be annotated with budget information.*

The models created to represent the system's structure need to contain budget information. This information can be added in several ways, but since some standard budget types are always present (such as CPU, memory, storage and networking), tags are a good way to accomplish this.

A stereotypes can contain an arbitrary number of tags. The stereotype presented in Chapter 6 does contain the tags for these budget types. Applying the stereotype to a UML model element in RSA automatically makes the associated tags available for that model element.

Whereas tags are the attributes of a stereotype, classes can still contain other attributes themselves. Extra budget information can easily be added by introducing new attributes to a class. This is all standard UML notation and is supported by the UML modeling capabilities of RSA.

Budget information has to be provided by the engineers. Correctness of this information is not checked by the tool and has to be ensured by the engineers.

**Requirement 4** *The tool has to evaluate constraints automatically.*

Constraints are specified as tags of «allocate» relationships and consist of expressions referring to budgets of the model elements related by the allocation. To evaluate a constraint and return a boolean result, the tool has to be able to parse and interpret an expression and calculate the value of referenced budgets. The parser generator ANTLR is used for this purpose [29].

Just as budget information, constraints have to be provided by the engineers, who have to ensure their correctness.

## 7.4 Use Cases

There are four basic tasks an engineer should be able to perform with the tool. They are derived directly from the purpose description of the tool, as illustrated in Figure 7.1.

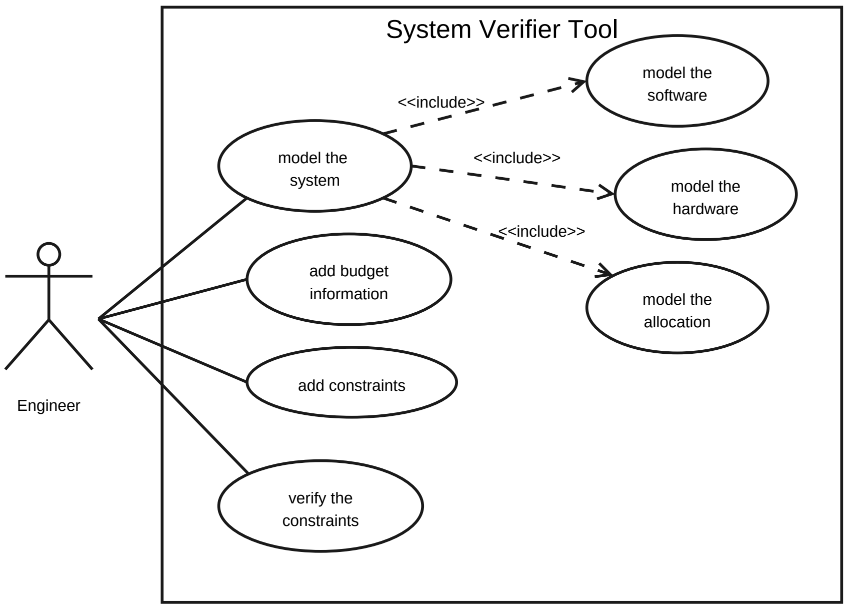


**Figure 7.1** — Use cases for the System Verifier Tool

The *model the system* use case consists of three use cases, which correspond to the sub-models of a system: software, hardware and allocation.

## 7.5 Implementation

The tool is implemented as two separate Eclipse plugins, *ModelStructure* and *SystemVerifier*. The implementation of both is discussed in the sections below.

### 7.5.1 ModelStructure plugin

The ModelStructure plugin basically reads a UML model created in RSA and constructs a corresponding Java model. Whereas the UML model consists of stereotyped classes, attributes, tags and relationships, the Java model consists of Java classes with attributes.

The Java classes of the ModelStructure plugin are arranged into four packages, all prefixed with `com.thales`: `modelStructure`, `modelStructure.expression`, `modelStructure.model` and `modelStructure.structure`. These packages are discussed in the sections below.

#### modelStructure

The `com.thales.modelStructure` package contains the basic classes of the plugin. It is shown in Figure 7.2

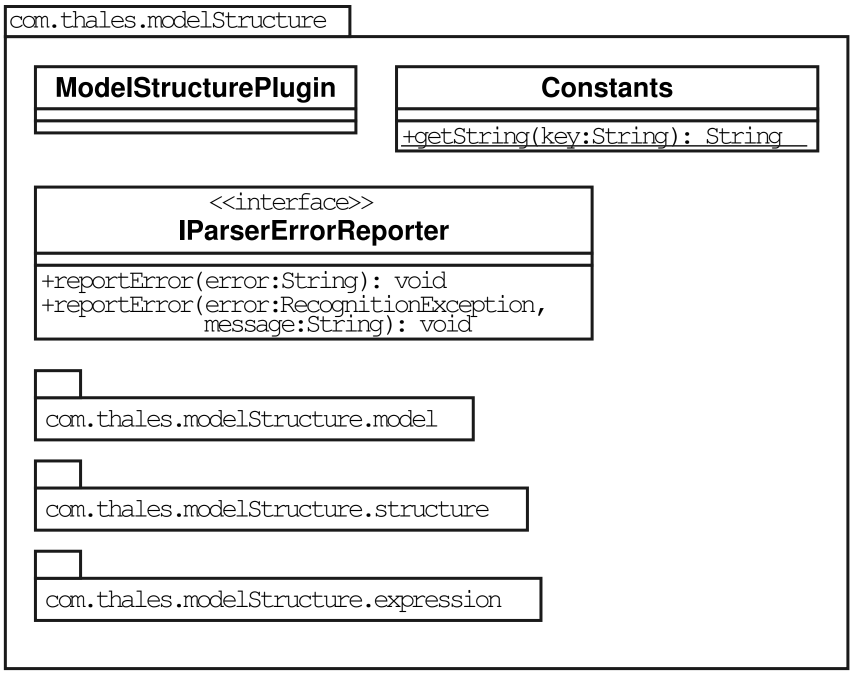

com.thales.modelStructure
ModelStructurePlugin
Constants
+getString(key:String): String
<<interface>>
IParserErrorReporter
+reportError(error:String): void
+reportError(error:RecognitionException, message:String): void
com.thales.modelStructure.model
com.thales.modelStructure.structure
com.thales.modelStructure.expression

**Figure 7.2 —** modelStructure package

- **ModelStructurePlugin** This is the base class of the plugin which extends the Eclipse `Plugin` class. `Plugin` is the abstract superclass of all plugin runtime class implementations. The class overrides the plugin's lifecycle methods `start(BundleContext)` and `stop(BundleContext)` [8].
- **Constants** The `Constants` class extracts all `String` constants from the classes of this plugin, so they can be used uniformly. This is especially useful for names of stereotypes and tags, because they are stored in one place only.

  As shown in Figure 7.2, the class only contains the `getString(key:String)` method which is used to retrieve the String corresponding to the provided key from the `constants.properties` file. This properties file contains a number of `key=value` pairs and eases the task of changing the names of stereotypes when necessary.
- **IParserErrorReporter** Classes implementing this interface have to implement its two methods for error reporting.

### modelStructure.expression

The `modelStructure.expression` package contains classes generated by the parser generator ANTLR [29]. The expressions specified as values of budgets need to be parsed and interpreted to determine their values. Parsing and interpretation are both performed by classes generated by ANTLR.

ANTLR first creates an abstract syntax tree (AST) of the input, based on the definition of the grammar in the `parser.g` file. The interpreter of the `interpreter.g` file specifies a treewalker for the AST created by the parser.

- **parser.g** Both a lexer and a parser are defined in this file. The lexer reads the input and converts it into tokens. The parser creates an AST based on these tokens. The grammars of the lexer and parser are listed in Appendix A.
- **interpreter.g** The interpreter is defined in this file as a treewalker. The grammar is listed in Appendix A.

Compiling the `.g` ANTLR source files results in a number of files to be created by ANTLR. The most important ones are the Java source files containing the lexer, parser and interpreter that can actually be compiled by the Java compiler.

### modelStructure.model

This package contains classes that are required to produce a model of Java objects from a UML model created in RSA. The allocations are contained in this package, as well as budgets.

- **Allocation** This class represents all allocation relationships between exactly two model elements, so it contains exactly one source and one target element. Furthermore, it keeps track of the ratios at all relationships and the constraints specified on those relationships. Based on the ratios, the allocation factor can be calculated, defining the contribution of the source element budgets to the target element.
- **AllocationModel** The AllocationModel contains all allocations and provides the functionality to *compact* allocation relationships between the same source and target elements. Compactification ensures that at most one Allocation instance remains per source – target element pair.
- **Budget** Each element has a budget, represented by this class. A budget has several properties, such as a name, value and unit and provides functions to calculate the value of the budget.
- **SystemModel** The SystemModel is the encompassing class, containing all information about the model, such as the structural model and allocations.
- **UMLModelParser** This is the main class for parsing UML models created in RSA. It takes a UML model (`org.eclipse.uml2.impl.ModelImpl`) as input and produces a SystemModel, containing all required properties of the UML model.

### modelStructure.structure

All Java representations of model elements are contained in this package. BudgetElement is the abstract base class for all model element classes, as shown in Figure 7.3.

- **BasicComponent** Class representing a component without any children. Part of the composite pattern.
- **Component** Base class for components and part of the composite pattern.

- **CompositeComponent** Class representing a component with children (other components). Also part of the composite pattern.
- **Delegate** Class representing a delegate. The class can only be owned by either a SoftwareModel or a HardwareModel element, depending on the contained elements.
- **HardwareModel** The base class for a hardware model. Instances of this class contain Nodes and Delegates.
- **Layer** Class representing a layer.
- **BudgetElement** Base class for all model elements. This class provides all basic functionality for a model element and contains all common properties, such as a name and a list of children.
- **Node** The class representing a node in the hardware system.
- **SoftwareModel** The encompassing class for a software model. It contains a number of layers and, possibly, delegates.

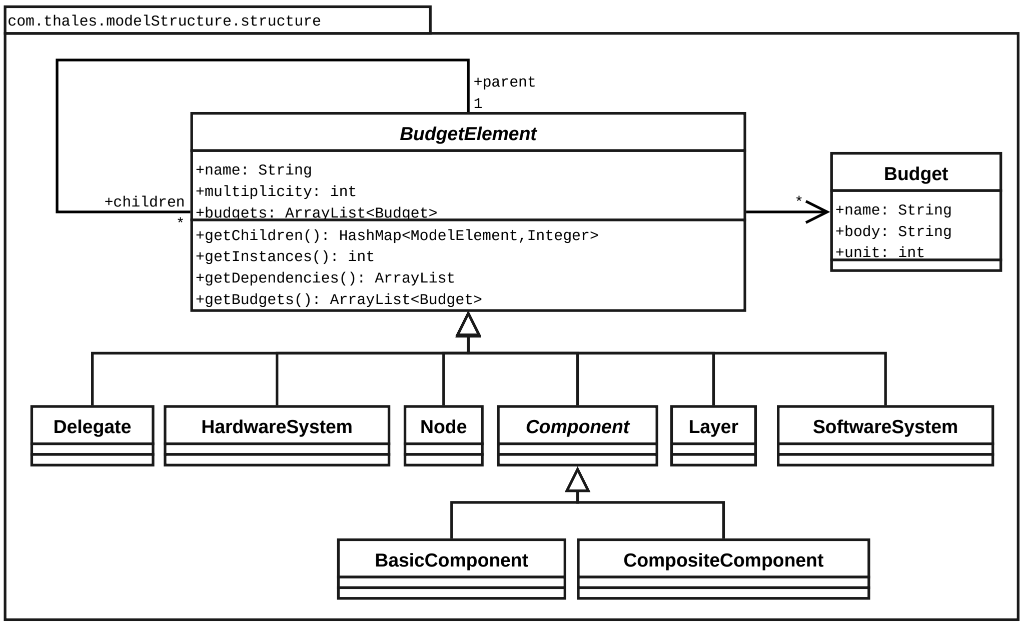


**Figure 7.3 —** modelStructure.structure package

The stereotypes of all model elements inherit from the common abstract stereotype «*TBudgetElement*». This abstract stereotype is reflected by the BudgetElement Java class. All other classes, representing the hardware and software systems, nodes, layers and components, are subclasses of the BudgetElement class. The Delegates have similar properties and also inherit from BudgetElement.

### 7.5.2 SystemVerifier plugin

Whereas the ModelStructure plugin is concerned with parsing UML models and creating a corresponding structure in Java classes, the SystemVerifier plugin uses the ModelStructure plugin to verify the associated constraints and display the results in a dedicated view.

This plugin thus consists of two main packages: `systemVerifier.engine`, an engine for the verification of constraints and `systemVerifier.views`, an eclipse extension providing a view for displaying the verification results. Furthermore, it contains the main class of the plugin.

- **SystemVerifierPlugin** Similarly to the ModelStructurePlugin class, the SystemVerifierPlugin extends the Eclipse `Plugin` class. Extending this class makes the SystemVerifierPlugin a true Eclipse plugin.

### systemVerifier.engine

Just like the calculation of budget expressions, constraint expressions need to be parsed and interpreted in order to evaluate them. Again, the ANTLR parser generator is used to create a lexer, parser and interpreter for the constraint expressions.

The ANTLR specifications of the parser and interpreter are also named `parser.g` and `interpreter.g` and closely resemble the specifications for the budget parser and interpreter. Their additional value is the addition of boolean operators. Furthermore, whereas the interpreter for budget expressions returns a numerical value, the interpreter for constraints returns a boolean value, indicating whether the constraint has been satisfied (`true`) or not (`false`).

Besides the classes generated by ANTLR, the verification engine consists of two more classes.

- **VerificationResult** The result of the evaluation of a constraint is stored in an instance of the VerificationResult class. Such a result naturally contains the boolean result value of the constraint, but also the values of the variables of the constraint expression. Displaying these values besides the result of the evaluation greatly eases the process of debugging a system when a constraint evaluates to false.

- **VerifierEngine** The VerifierEngine retrieves all constraints from the model created by the ModelStructure plugin and feeds them to the parser and interpreter. Interpretation yields one VerificationResult per constraint, which are stored in the VerifierEngine class to be retrieved by a view and displayed to the engineer.

### systemVerifier.views

The views package contains the following classes, which create the view and ensure that the proper information is displayed. A View is a vital part of the Eclipse framework and consists of a frame within the IDE which can contain any kind of information. Information is provided to the view via an implementation of the MVC pattern in Eclipse.

- **ConstraintTreeContentProvider** The content provider for the table showing the constraints and variable values. The class gets an object as input and returns an array of objects to be shown in the table. These are the top-level objects. Furthermore, the class defines the children to be shown for each object.

- **ConstraintTreeLabelProvider** The label provider defines what needs to be shown for each object in the table of constraints. Depending on the colomn, this class defines the text to be shown in the corresponding table cell.

- **StructureTreeContentProvider** This class is the content provider for the table showing the structure of the model and all budgets. It defines that the hardware model and the software model elements are to be shown as the top-level elements in the table and uses the structure to derive the children of those elements.

- **StructureTreeLabelProvider** The label provider for the structure table defines what needs to be put in the cells of the table. Depending on the column, this class produces the text to be shown in the table.

- **StructureTreeSorter** Elements in the structure table are sorted alphabetically in an ascending order by default. This class ensures that the elements appear in a sorted order in the table.

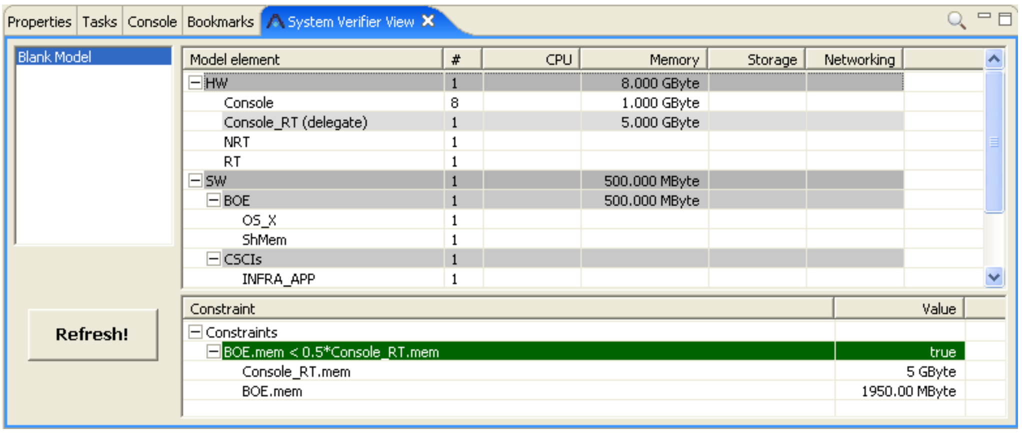
- **SystemVerifierView** The main class for the view. This class extends the Eclipse `ViewPart` class, which is required for any view. Furthermore, it implements the IParserErrorReporter interface, allowing it to receive error messages from the UML parser. The SystemVerifierView creates all parts of the view, as shown in Figure 7.4.

  The view consists of four parts. The list on the top-left of the view shows all opened UML models. Figure 7.4 shows that two models are open in RSA: *Blank Model* and *Blank Model1*.

  The table on the top-right of the view shows the structure of the UML model and assigned budgets. The background of a row is colored according to the decomposition hierarchy level of the model element it contains. A darker background implies a higher level in the decomposition hierarchy. The background of a row can also be colored *orange*. An orange row indicates a warning to the engineer that one of the budgets of the corresponding model element is smaller than the sum of its children. The larger summation of the children's budgets is used as *the* budget for the model element instead of the budget defined for that model element.

  The table on the bottom-right of the view shows all constraints and their contained variables. The background of each constraint is either colored red or green, depending on the result of the evaluation. Values of the variables are shown below the constraints.

  The button on the bottom-left of the view causes the entire view to be refreshed. The list of open UML models is refreshed, the selected UML model is parsed and budgets are recalculated. All constraints are evaluated and evaluation results are shown in the view.



| Model element | # | CPU | Memory | Storage | Networking |
|---|---|---|---|---|---|
| ⊟ HW | 1 | | 8.000 GByte | | |
| Console | 8 | | 1.000 GByte | | |
| Console_RT (delegate) | 1 | | 5.000 GByte | | |
| NRT | 1 | | | | |
| RT | 1 | | | | |
| ⊟ SW | 1 | | 500.000 MByte | | |
| ⊟ BOE | 1 | | 500.000 MByte | | |
| OS_X | 1 | | | | |
| ShMem | 1 | | | | |
| ⊟ CSCIs | 1 | | | | |
| INFRA_APP | 1 | | | | |

| Constraint | Value |
|---|---|
| ⊟ Constraints | |
| ⊟ BOE.mem < 0.5*Console_RT.mem | true |
| Console_RT.mem | 5 GByte |
| BOE.mem | 1950.00 MByte |

**Figure 7.4** — System Verifier View

# Chapter 8

# Conclusion

This chapter concludes the thesis. The achieved results are discussed briefly and evaluated. Finally, a brief discussion on related work and future work is given.

## 8.1 Evaluation

The main objective of this research was to develop a method for specifying and verifying performance budgets in an early design phase for software intensive systems. To solve this problem, the following three parts of the problem statement have been fulfilled.

#### Definition of a method for the specification of performance budgets.

To specify performance budgets for a system, it is generally required that a model of the system exists to which this information can be added. The UML model we have developed can be used to define the structure of the example combat management system, but also of other systems exhibiting a similar structural hierarchy. All basic elements of a system (SoftwareSystem, Layer, Component for software and HardwareSystem and Node for hardware) are available as UML stereotypes.

With our model performance budgets can be added to all elements of the created structural description of a system. Budgets denote the amount of resources offered by hardware or demanded by software. The four budget types CPU, memory, storage and networking are by default available for all model elements. An unlimited number of additional budgets and parameters can however be defined for each element of the model of a system.

Budgets are not required to have fixed values, but can depend on other budgets. A budget is dependent when its value is not a constant, but contains a reference to another budget. Dependencies between components can thus be specified using dependent budgets. Values of budgets are specified using our own expression-based language.

The division of a software system into layers and components is widely applied nowadays, making the model we have developed applicable to a wide range of systems.

#### Creation of a system independent framework for the expression of constraints to verify performance budgets.

Constraints define restrictions on the resource utilization. To define in our model which software parts run on which hardware parts, allocation relationships can be defined between any pair of a software element and a hardware element. It is on these allocation relationships where resource utilization is defined, because they relate software to hardware. The relations between budgets of software and hardware, the constraints, are thus best specified on these allocation relationships.

In our model, constraints are independent of a hardware or software system, because they can be specified on every allocation relationship to relate any pair of budgets. Since allocation relationships can be defined for every combination of a software element and a hardware element, constraints can cover the complete boundary between software and hardware. Budgets of all types can be constrained, thus providing engineers with maximum freedom. Just as budgets, constraints are specified using our expression-based language, which allows the application of all basic arithmetic and logic operations.

Development of an engine which calculates the resource utilization of a system.

The described model has been created as a set of UML stereotypes, which are packaged in the TProfile UML profile. This profile has been implemented in RSA and, when applied, provides engineers with all model elements required to create a structural model of a system, add budget information and define allocation relationships and constraints.

The System Verifier Tool has been developed as a plugin for RSA and integrates with the IDE. It contains the TProfile and can read the models of systems created using the TProfile by extracting all relevant information. The System Verifier Tool calculates budget values, which depend on the allocation relationships. It does this by using the parser generator ANTLR, which builds an abstract syntax tree of an expression and evaluates it using the tree. Subsequently, constraints are parsed and evaluated, using ANTLR as well, and the result of each constraint is displayed.

Resource utilization is calculated by our System Verifier Tool when budget values are calculated. Combined with the allocation relationships, the budget values themselves define the resource utilization for a system. Evaluation of constraints presents engineers with additional information and allows them to experiment with budget values (hence resource utilization as well) and restrictions.

## 8.2 Discussion

Although the application of our method method can help Thales' engineers to determine whether a complex software system fits on a distributed hardware system, it cannot replace the knowledge and experience of systems engineers. The developed System Verifier Tool aids engineers in determining resource utilization in an early development phase, but still requires the input of budget information and constraints. These are not always exactly known and sometimes have to be estimated. The quality of estimations, as well as the correctness of the structural model of a system, relies on the knowledge and experience of engineers and is not verified by the tool. The System Verifier Tool can however be used to evaluate a number of different structures, budget values and constraints and thus support the estimations in an early development phase.

The model created during the research presented in this thesis has been developed to capture properties of combat management systems, but can be used for other kinds of systems as well. The hierarchical structure of software and the distributed nature of the hardware system are not unique to combat management systems, but also fit other types of systems. The System Verifier Tool can therefore be used to evaluate performance properties of those systems as well and thus not only provides a contribution to the systems development process of Thales, but to systems performance research in general.

The use of SysML as a basis for our model eases the process of integrating this model into the systems development process. The UML model closely resembles its SysML counterpart and can easily be converted to SysML. However, this is not feasible without proper support for SysML in RSA. Existing support for UML in RSA is better, but not perfect, which is why objects cannot be used. This imperfect support is however also taken advantage of, because it makes the use of Real values for association multiplicities possible.

Originally, constraints were to be specified using OCL. This idea was not implemented because of the same reason SysML was not used; the lack of support in RSA. The current implementation which makes use of ANTLR generated parsers and interpreters is not easily replaced by an implementation which makes use of OCL. An advantage of our approach is that every syntactical construction is possible, because it can be defined in the ANTLR source files. Inclusion of OCL would however introduce a standardized syntax and integrate easily with UML models.

## 8.3 Related Work

Most research on performance engineering focuses on process scheduling, communication latencies, response times and throughput measurements. A lot of effort is put into standardizing these types of performance evaluation, for instance by means of the SPE research by Smith et al. [34, 35, 36] and by the SPT profile of the OMG [24].

The methods proposed by those research areas can only be applied when the structure of hardware and software has already been defined. If application of our method already indicates that a hardware system cannot provide enough resources for a software system, it is pointless to apply more fine-grained performance engineering methods such as throughput measurements, because it is very likely that they will indicate that the system performance is unsatisfactory. Application of those methods does not make sense, when the System Verifier Tool already indicates that the resource demands a software system imposes on a hardware system are too large.

The research reported in this thesis focuses on performance engineering in an earlier phase of systems development than the above mentioned research. Also, it is not about the dynamic behavior of software, but about the static structure of software and relations to the hardware it runs on. The main contribution of this research, not only to Thales, but also to the research area of performance evaluation, is therefore that the presented method can already be applied from the moment the first system decomposition specification is being made and one can continue apply the method until the system structure is satisfactory.

## 8.4 Future Work

This research covers the first of three performance engineering steps for Thales. The following step is concerned with determining the most important functional flows in a system and checking quality of service aspects of systems. The System Verifier Tool built during this research can be used as a basis for tool support of the future work. Special attention should be paid to the following aspects of this research:

### Budgets as Classes

Currently, budget information is expressed by means of attributes of the classes representing the model elements. Default budgets are introduced by the tags of applied stereotypes. Even though this allows engineers to express a variety of budgets and other information, adding extra information to a budget is not easily done. Representation of budgets by classes would give the model an extra degree of freedom, because extra budget information can be added easily. This is illustrated in Figure 8.1.

An advantage of representing a budget by an attribute is that it reduces the size of a model. Adding an extra instance for each new budget takes up space in the model and can make it overly complex. The System Verifier Tool however, does implement a budget as a Java class. Changing the model to include UML classes for budgets as well will therefore have little consequence for the implementation of the system verifier.

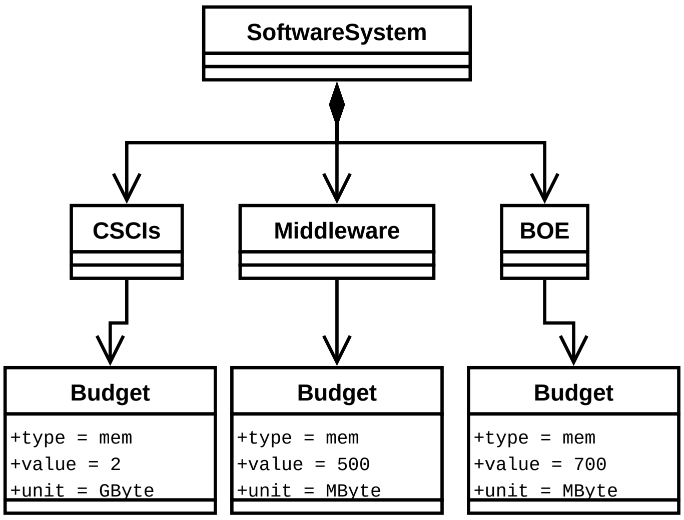


**Figure 8.1** — Budgets represented by class instances.

### Allocation Relationship Dependencies

Allocation relationships are independent of each other. This is not always desirable, because allocation relationships influence the values of budgets. Consider a budget of a software component which is allocated to a hardware node and the complete software system which is allocated to the hardware system, as illustrated in Figure 8.2.

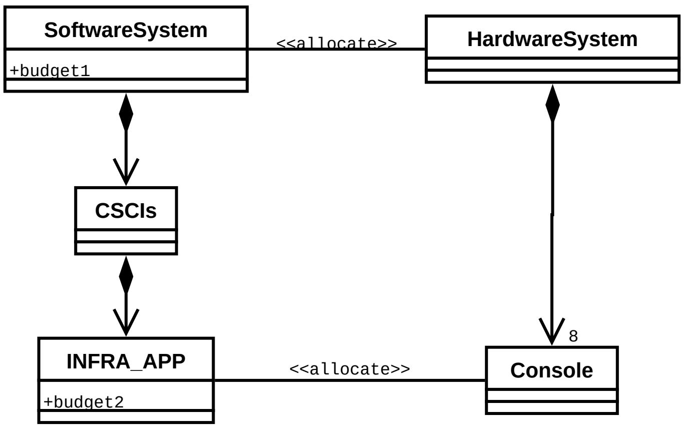


**Figure 8.2** — Allocation dependency influences budget values.

The contribution of budget2 is determined by the bottom allocation relationship. The value of budget1, which depends on budget2, does currently *not* use the value of budget2 as influenced by the allocation relationship, but the original value which was specified by the engineer.

It is recommended that such a dependency will be implemented in a continuation of this research, because it is more intuitive to the user of the System Verifier Tool.

### SysML

The UML models of systems presented in this thesis are all based on SysML models. The only reason to use the UML is the lack of support for SysML in RSA. However, the final version of SysML has only recently been released, so tool support is likely to improve.

With better tool support for SysML, it is not difficult to convert the presented UML models to SysML models. The structure of software and hardware system is expressed by class diagrams, which correspond directly to SysML block definition diagrams. Budgets can be expressed in specialized compartments of these blocks and allocation can, just as with the UML, be expressed by allocation relationships. Only constraints are best expressed differently in SysML, because SysML introduces constraint blocks and the corresponding parametric diagram, which are well suited to define constraints. SysML constraint blocks support custom languages, so the expression-based language we have defined can still be used.

The System Verifier Tool does not require many changes as well. Only the parser, which constructs a Java based model from a UML model, needs to be modified to take SysML models as input.

Altogether, a shift from UML to SysML can be done rather easily. SysML can improve communication, because it supports the modeling of software as well as hardware and more. If possible, when the presented research is continued, we recommend to take SysML into consideration.

A lot of research has already been done regarding functional flows and quality of service aspects of systems. Especially the SPE research of Smith et al. [34, 35, 36] is likely to be a very interesting source of information for Thales. However, if application of the method presented in this research indicates that the performance of a system is unsatisfactory, evaluation of performance properties such as latencies and throughput is less likely to succeed.

# Bibliography

[1] Alain Abran and James W. Moore. *SWEBOK, Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society Professional Practices Committee, 2004.

[2] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[3] Azad Bolour. *Notes on the Eclipse Plug-in Architecture*. Bolour Computing, July 2003. Eclipse Corner Article.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented software architecture. A system of patterns*. John Wiley & Sons Ltd., Chichester, England, July 2001.

[5] R. Byrne. Managing Complexity: An Approach to Net-Centric Ops. presentation, September 2005. MITRE Corporation.

[6] Phillippe Desfray. UML Profiles versus Metamodel extensions: An ongoing debate. `http://www.softeam.fr`, 2000. Softeam.

[7] The Eclipse project. `http://www.eclipse.org`, 2006.

[8] Eclipse Platform API Specification. Release 3.0.

[9] EmbeddedPlus Engineering, LLC. SysML Toolkit for the IBM Rational Software Development Platform (RSDP). `http://www.embeddedplus.com`, 2006.

[10] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics, A Rigorous & Practical Approach*. PWS Publishing Company, Boston, second edition, 1997.

[11] Sanford Friedenthal, Alan Moore, and Rick Steiner. OMG Systems Modeling Language Tutorial. INternational COuncil on Systems Engineering (INCOSE), July 2006.

[12] J.H. van 't Hag. "Data-centric to the Max", The SPLICE architecture experience. Thales Naval Nederland, February 2003.

[13] Cecilia Haskins, editor. *Systems Engineering Handbook*. Number TP-2003-002-03 in INCOSE. International Council on Systems Engineering (INCOSE), June 2006. Version 3.

[14] Ivo ter Horst. DESIDE, DEcision-support using System sImulation in a Design Environment. Internship report, University of Twente and Thales Naval Nederland, May 2006.

[15] IBM. Rational Software Architect. `http://www.ibm.com/software/awdtools/architect/swarchitect/`, 2006.

[16] International Organization for Standardization. Software engineering - Product quality - Part 1: Quality model. ISO/IEC standard 9126-1, ISO/IEC, 2001. `http://www.iso.org`.

[17] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained.* object technology series. Addison-Wesley, April 2003.

[18] Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engineering.* McGraw-Hill, 2001.

[19] Object Management Group (OMG). `http://www.omg.org`.

[20] Klasse Objecten. OCTOPUS: OCL Tool for Precise Uml Specifications. `http://www.klasse.nl/octopus`, 2006.

[21] OMG. MDA Guide. version 1.0.1 omg/2003-06-01, Object Management Group, June 2003.

[22] OMG. Data Distribution Service for Real-Time Systems Specification. Technical Report formal/05-12-04, Object Management Group, December 2005.

[23] OMG. OCL 2.0 Specification. Technical Report ptc/2005-06-06, Object Management Group, June 2005.

[24] OMG. UML Profile for Schedulability, Performance, and Time Specification. version 1.1 formal/05-01-02, Object Management Group, January 2005.

[25] OMG. Unified Modeling Language: Superstructure. version 2.0 formal/05-07-04, Object Management Group, August 2005.

[26] OMG. Meta Object Facility (MOF) Core Specification. version 2.0 formal/06-01-01, Object Management Group, January 2006.

[27] OMG. Systems Modeling Language (SysML) Specification. Final Adopted Specification ptc/06-05-04, Object Management Group, May 2006.

[28] OMG. Unified Modeling Language: Infrastructure. version 2.0 formal/05-07-05, Object Management Group, March 2006.

[29] Terence Parr. ANTLR, ANother Tool for Language Recognition. `http://www.antlr.org`, 2006.

[30] Jorge Enrique Pérez-Martínez. Heavyweight extensions to the uml metamodel to describe the c3 architectural style. *SIGSOFT Softw. Eng. Notes*, 28(3):5–5, 2003.

[31] The Eclipse project. UML2 UML. `http://www.eclipse.org/uml2/?project=uml2#uml2`, 2006.

[32] David S. Rosenblum. Lightweight Extension Mechanisms for UML. Lecture notes Advanced Analysis and Design (GS02/4022), 2005. University College London.

[33] Community site for metamodeling and semantic modeling. What is metamodeling, and what is it good for? `http://www.metamodel.com`, September 2006.

[34] Connie U. Smith. Origins of Software Performance Engineering: Highlights and Outstanding Problems. In R. Dumke et al., editor, *Performance Engineering*, LNCS 2047, pages 96–118. Springer-Verlag, Berlin Heidelberg, 2001.

[35] Connie U. Smith and Lloyd G. Williams. Software Performance Engineering. In Luciano Lavagno, Grant Martin, and Bran Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*. Kluwer, 2003.

[36] Connie U. Smith and Murray Woodside. Performance Validation at Early Stages of Software Development. In E. Gelenbe, editor, *System Performance Evaluation: Methodologies and Applications*. CRC Press, 1999.

[37] Standard Performance Evaluation Corporation. `http://www.spec.org`, 2006.

# List of Figures

# List of Tables

# Appendix A

# Expression Syntax

This appendix describes the exact syntax of expressions for budget values and constraints. Each of the following sections shows the source code from which the ANTLR compiler creates a lexer, parser and interpreter respectively. These generated Java source files can easily be integrated into any application.

## A.1 Lexer Grammar

The lexer creates a stream of tokens from a stream of characters (an expression) it receives as input. The lexer tries to match all characters and create tokens. A '(' for example is assigned the 'LPAREN' token.

The listing below shows the ANTLR grammar for the lexer. It consists of a list of tokens and the characters to be matched.

Line 58 shows the definition for identifiers: `(LOWER|UPPER|'_')(LOWER|UPPER|'_' |DIGIT)`. The rule defines that an identifier can consist of lowercase and uppercase letters, underscores and digits, but cannot start with a digit.

**Listing A.1 —** Antlr source of the Lexer

```
   class ConstraintLexer extends Lexer;

3  options{
     k=2;
     charVocabulary='\u0003'..'\u7FFE';
     testLiterals = false;
     exportVocab = Constraint;
8  }

   LPAREN        : '(' ;
   RPAREN        : ')' ;
   EQUAL         : "==" ;
13 LNOT          : '!' ;
   NOT_EQUAL     : "!=" ;
   DIV           : '/' ;
   PLUS          : '+' ;
   MINUS         : '-' ;
18 STAR          : '*' ;
   GE            : ">=" ;
   GT            : '>' ;
   LE            : "<=" ;
   LT            : '<' ;
23 LOR           : "||" ;
   LAND          : "&&" ;
   QUESTION      : '?' ;
   COMMA         : ',' ;
   COLON         : ':' ;
28 ASSIGN        : '=' ;
```

```
PLUS_ASSIGN   : "+=" ;
MINUS_ASSIGN  : "-=" ;
STAR_ASSIGN   : "*=" ;
DIV_ASSIGN    : "/=" ;
DOT           : '.' ;

WS  : ('␣'
    | '\t'
    | '\n'
    | '\r')
      { _ttype = Token.SKIP; }
    ;

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

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

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

IDENTIFIER
  options {
    testLiterals=true;
  }
  : (LOWER|UPPER|'_') (LOWER|UPPER|'_'|DIGIT)*
  ;

NUMBER
  : (DIGIT)+ ((DOT | COMMA) (DIGIT)+)?
  ;
```

## A.2 Parser Grammar

Similar parsers are used for the parsers of budget value expressions and constraint expressions. The grammar of Listing A.2 has its first rule named *constraint*, but *expression* is applicable as well. Please note that the `reportError(...)` method on line 23 needs to be in the definition of the parser to catch the exceptions generated during parsing.

The parser takes a stream of tokens, generated by the lexer, and produces an Abstract Syntax Tree (AST). The rules of the parser all generate specific nodes in the tree. The `additiveExpr` rule on line 58 for example, creates either a `PLUS` or a `MINUS` node (depending on the received token) with two children representing the operands of the addition or subtraction.

**Listing A.2 —** ANTLR source of the Parser

```
    header{
 2  package com.thales.systemVerifier.engine;

    import com.thales.modelStructure.IParserErrorReporter;
    }

 7  class ConstraintParser extends Parser;

    options {
      exportVocab = Constraint;
      k=1;
12    buildAST = true;  // uses CommonAST by default
      defaultErrorHandler = false;
    }

    tokens{
17    CONSTRAINT_AST; UNARY_MINUS; UNARY_PLUS; BOOL_TRUE; BOOL_FALSE; TYPE;
    }
```

```
{
  private IParserErrorReporter reporter;

  public void reportError(RecognitionException ex) {
    reporter.reportError(ex, "");
  }

  public void setErrorReporter(IParserErrorReporter reporter){
        this.reporter = reporter;
  }
}

constraint
  : exprList EOF!
    {#constraint = #([CONSTRAINT_AST, "constraint"], #constraint) ; }
  ;

exprList
  : logicalOrExpr (COMMA! logicalOrExpr)*
  ;

logicalOrExpr
  : logicalAndExpr (LOR^ logicalAndExpr)*
  ;


logicalAndExpr
  : equalityExpr (LAND^ equalityExpr)*
  ;

equalityExpr
  : relationalExpr ((NOT_EQUAL^ | EQUAL^) relationalExpr)*
  ;

relationalExpr
  : additiveExpr ((LT^ | GT^ | LE^ | GE^) additiveExpr)*
  ;

additiveExpr
  : multiplicativeExpr ((PLUS^ | MINUS^) multiplicativeExpr)*
  ;

multiplicativeExpr
  : unaryExpr ((STAR^ | DIV^ | MOD^) unaryExpr)*
  ;

unaryExpr
  : MINUS^ {#MINUS.setType(UNARY_MINUS);} unaryExpr
  | PLUS^ {#PLUS.setType(UNARY_PLUS);} unaryExpr
  | unaryExpressionNotPlusMinus
  ;

unaryExpressionNotPlusMinus
  : LNOT^ primaryExpr
  | primaryExpr
  ;

primaryExpr
  : LPAREN! logicalOrExpr RPAREN!
  | constant
  ;

constant
  : NUMBER^ (type:IDENTIFIER
    {
      #type.setType(TYPE);
    }
    )?
  | i:IDENTIFIER
    {
      if(i.getText().equals("true")) #constant.setType(BOOL_TRUE);
      else if(i.getText().equals("false")) #constant.setType(BOOL_FALSE);
    }(DOT^ IDENTIFIER)?
  ;
```

## A.3 Interpreter Grammar

Interpreters take an AST (generated by the parser) as input and 'walk' the AST to generate output. The rules correspond to the nodes, created by the parser. There are differences between the two treewalkers of the budget value interpreter and the interpreter for constraint expressions. Budget values never have a boolean value, while constraint expressions never yield a non-boolean value.

**Listing A.3 —** ANTLR source of the interpreter

```
class ConstraintInterpreter extends TreeParser;

options {
  importVocab = Constraint;
}

constraint returns [boolean res]
{
  res = false;
}
  :   #( CONSTRAINT_AST res=boolExpr)
  ;

numExpr returns [Budget res]
{
  Budget p1, p2, min1, min2, s1, s2, d1, d2, mod1, mod2, q2, q3, um;
  boolean q1;
  res = null;
}
  : #(PLUS p1=numExpr p2=numExpr) { res = p1.add(p2); }
  | #(MINUS min1=numExpr min2=numExpr) { res = min1.min(min2); }
  | #(STAR s1=numExpr s2=numExpr) { res = s1.multiply(s2); }
  | #(DIV d1=numExpr d2=numExpr) {res = d1.divide(d2); }
  | #(UNARY_MINUS um=numExpr) {res = um.negate(); }
  | #(UNARY_PLUS res=numExpr) // simply return the expression
  | #(n:NUMBER (t:TYPE)?) { ... }
  | id:IDENTIFIER { ... }
  | #(DOT elName:IDENTIFIER budName:IDENTIFIER) { ... }
  ;

boolExpr returns [boolean res]
{
  res = false;
  Budget gt1, gt2, lt1, lt2, ge1, ge2, le1, le2, eq1, eq2, ne1, ne2;
  boolean and1, and2, or1, or2, q1, q2, q3, not;
}
  : BOOL_TRUE { res = true; }
  | BOOL_FALSE { res = false; }
  | #(GT gt1=numExpr gt2=numExpr) {res = gt1.greaterThan(gt2); }
  | #(LT lt1=numExpr lt2=numExpr) {res = lt1.lessThan(lt2); }
  | #(GE ge1=numExpr ge2=numExpr) {res = ge1.greaterEqualThan(ge2); }
  | #(LE le1=numExpr le2=numExpr) {res = le1.lessEqualThan(le2); }
  | #(EQUAL eq1=numExpr eq2=numExpr) {res = eq1.equal(eq2); }
  | #(NOT_EQUAL ne1=numExpr ne2=numExpr) {res = ne1.notEqual(ne2); }
  | #(LAND and1=boolExpr and2=boolExpr) {res = and1 && and2; }
  | #(LOR or1=boolExpr or2=boolExpr) {res = or1 || or2; }
  | #(LNOT not=boolExpr) {res = !not; }
  ;
```

# Appendix B

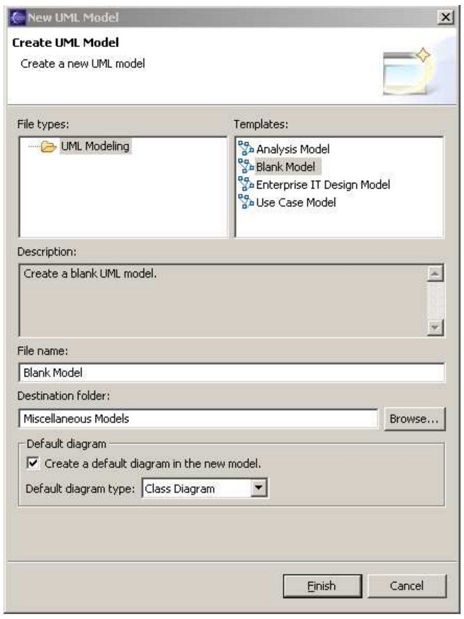
# System Verifier User Guide

This appendix can be used as a guideline to using the System Verifier Tool, which is a Rational Software Architect plugin. All modeling and verification steps are described and illustrated by screenshots of RSA. Following the steps in order results in the model of a very simple system with budgets and constraints that can be verified.

## B.1 Structure

First of all, a new model has to be created in RSA which can contain the model elements. Figure B.1 shows the screen with which a new UML model can be created. The default diagram type should be a *Class diagram*.

User actions: The screen can be accessed via the *File* → *New* menu and choosing the *UML Model* option.


New UML Model

Create UML Model

Create a new UML model

File types: UML Modeling

Templates: Analysis Model, Blank Model, Enterprise IT Design Model, Use Case Model

Description: Create a blank UML model.

File name: Blank Model

Destination folder: Miscellaneous Models

Default diagram

☑ Create a default diagram in the new model.

Default diagram type: Class Diagram

**Figure B.1 —** Create a new UML model.

More UML models can be created to contain multiple system models. A system, consisting of both a hardware and a software part is contained in at most one UML model. To compare several systems, multiple UML models have to be created.

## B.1.1 Model Elements

Model elements are all stereotyped UML classes and need therefore be created in Class Diagrams. Figure B.2 illustrates the creation of a new model element in a class diagram.

User actions: When hovering the mouse pointer over the modeling area, a menu appears from which new model elements can be created. The third icon can be used to create stereotyped classes. In this case, a «THWSystem» Class is created. The name of a model element can be changed by clicking on it and entering a new name.

Other model elements are also listed in the pop-up menu and can be created in the same way.

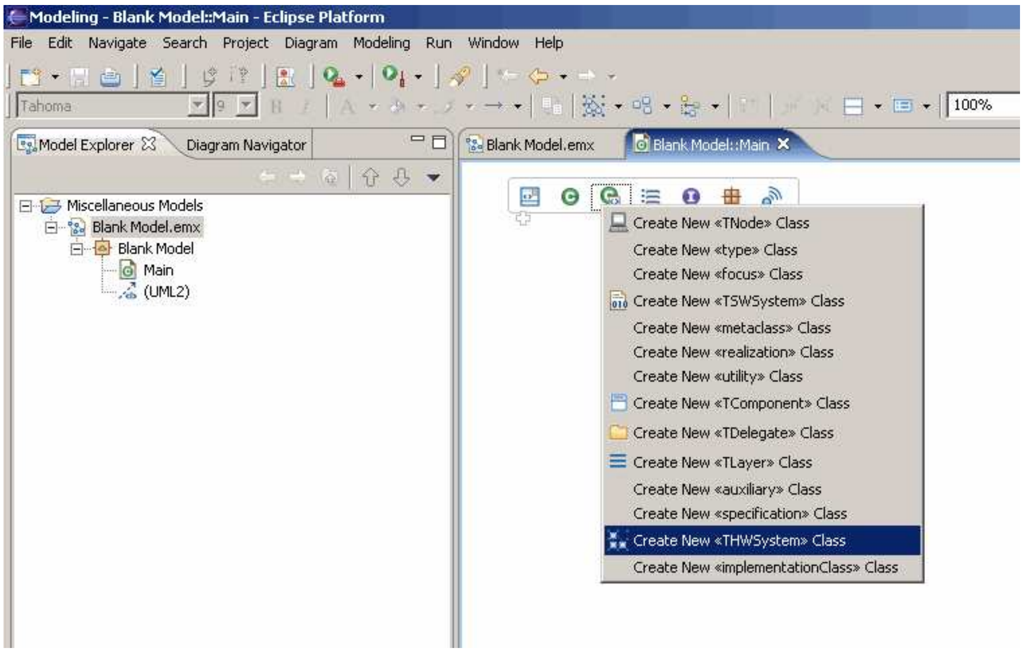

Modeling - Blank Model::Main - Eclipse Platform
File Edit Navigate Search Project Diagram Modeling Run Window Help
Tahoma
100%
Model Explorer
Diagram Navigator
Miscellaneous Models
Blank Model.emx
Blank Model
Main
(UML2)
Blank Model.emx
Blank Model::Main
Create New «TNode» Class
Create New «type» Class
Create New «focus» Class
Create New «TSWSystem» Class
Create New «metaclass» Class
Create New «realization» Class
Create New «utility» Class
Create New «TComponent» Class
Create New «TDelegate» Class
Create New «TLayer» Class
Create New «auxiliary» Class
Create New «specification» Class
Create New «THWSystem» Class
Create New «implementationClass» Class

**Figure B.2 —** Create a new stereotyped UML model element.

Sub-elements of a model element can be created as shown in Figure B.3.

User actions: When hovering the mouse pointer over a model element, the appearing handles can be used to create a relationship (in this case a Composition Association) to a new UML model element.

All nodes of which the hardware system is composed can be drawn this way. The composition association is created immediately and links the nodes to the hardware system.

Multiplicities provide a way to define the number of instances of a model element with respect to its parent. Figure B.4 shows that the multiplicity of the composition association to Console has been set to 8.

User actions: A multiplicity can be changed by clicking on the character denoting the multiplicity (a '*' by default) and entering a new value.

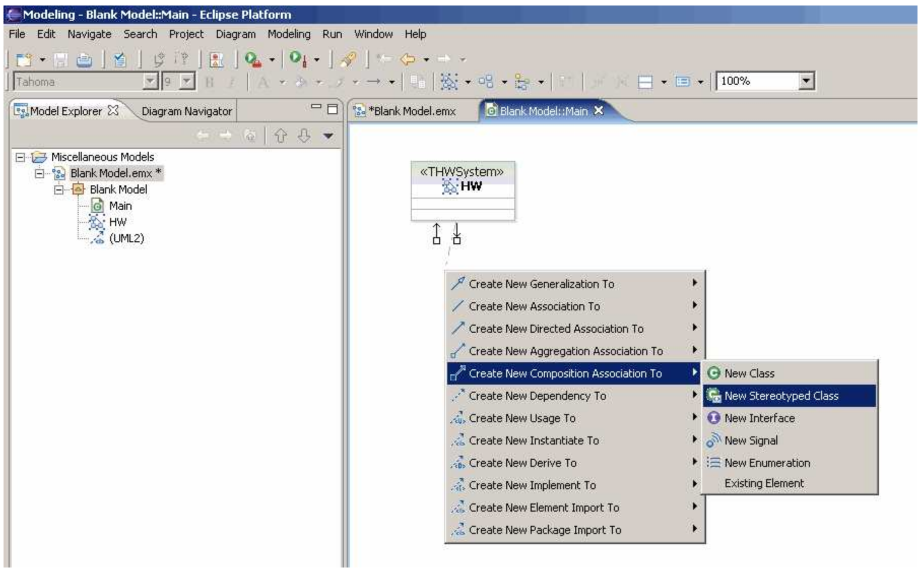

Modeling - Blank Model::Main - Eclipse Platform
File Edit Navigate Search Project Diagram Modeling Run Window Help
Model Explorer
Diagram Navigator
*Blank Model.emx
Blank Model::Main
Miscellaneous Models
Blank Model.emx *
Blank Model
Main
HW
(UML2)
«THWSystem»
HW
Create New Generalization To
Create New Association To
Create New Directed Association To
Create New Aggregation Association To
Create New Composition Association To
Create New Dependency To
Create New Usage To
Create New Instantiate To
Create New Derive To
Create New Implement To
Create New Element Import To
Create New Package Import To
New Class
New Stereotyped Class
New Interface
New Signal
New Enumeration
Existing Element

**Figure B.3 —** Create a new stereotyped UML model element as a child of another element.

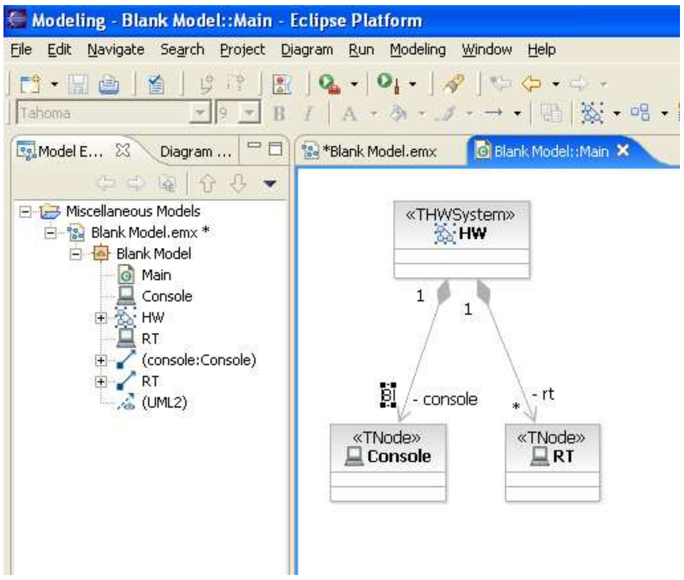


**Figure B.4 —** Multiplicities of association relationships can be changed. The composition association to Console now has a multiplicity of 8.

The default multiplicity of '*' is interpreted by the System Verifier Tool as a multiplicity of 1. Engineers do therefore not need to change the default '*' every time a multiplicity of 1 is required.

Names of relationships are not used in this model and can therefore be omitted from the diagram. Figure B.5 shows the menu presented to the engineer when right-clicking on the name of the relationship.

User actions: The name of the relationship can be removed by right-clicking on it and choosing *Delete from Diagram*.

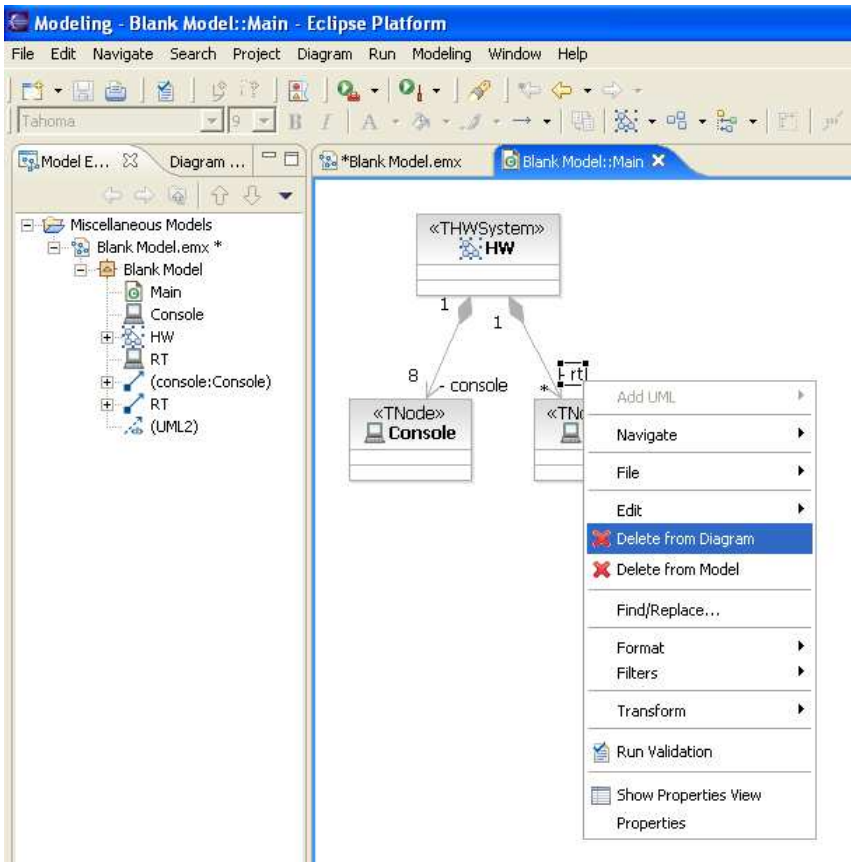


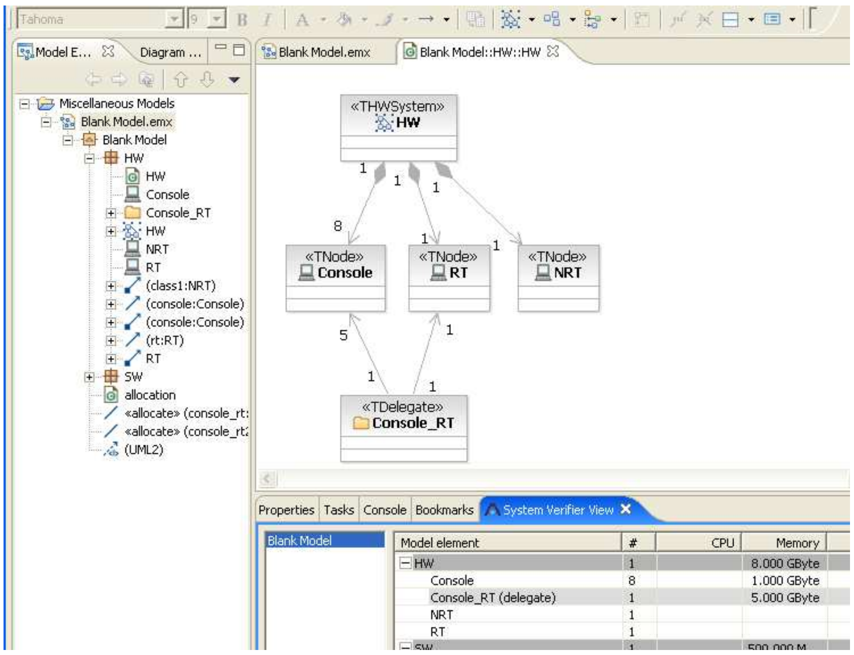
**Figure B.5 —** Names of associations are not required in the models and can therefore be deleted from the *diagram*.

Please note that deleting the name from the *diagram* has no influence on the relationship itself, but only on the diagram. Deleting the name from the *model* results in deleting the relationship from the model altogether!

### B.1.2 Delegates

Delegates provide a way to group model elements without influencing the model itself. Figure B.6 shows the creation of a new «TDelegate» element, grouping 5 Console nodes and 1 RT node.

User actions: Delegates can be created in the same way as other models elements, such as the «THWSystem» element (as shown in Figure B.2).



| Model element | # | CPU | Memory |
|---|---|---|---|
| HW | 1 | | 8.000 GByte |
| Console | 8 | | 1.000 GByte |
| Console_RT (delegate) | 1 | | 5.000 GByte |
| NRT | 1 | | |
| RT | 1 | | |
| SW | 1 | | 500.000 M |

**Figure B.6 —** Delegates are created in the same way as other model elements.

Please note that, in contrast to other structural model elements, delegates do *not* group other elements by means of composition relationships, but by *Directed Associations*. The multiplicities at the directed associations define how many instances are grouped by the delegate. In this example, the delegate groups 5 Console nodes and the RT node. Note that no more than the maximum number of instances (in this case 8 for the Console nodes) can be included in a delegate.

## B.1.3 Packages

Packages provide a way to organize model elements within one model. They prevent the model from becoming one big list of model elements and relationships in the Model Explorer, as shown in Figure B.7. A package creates a new branch in the Model Explorer tree and can be collapsed to hide the contained elements.

User actions: A new package can be added by right-clicking on the model and selecting *Add UML → Package*. In Figure B.7 a package called *HW* has already been created. Create another one called *SW* to contain the elements of the software system.

New elements can be created in a package, or existing elements can be moved to a package by dragging them to it.

User actions: Select all model elements of the Blank Model (including the *main* diagram and all relationships, but excluding the newly created packages) and drag them to the HW package. The elements are now contained in the package.

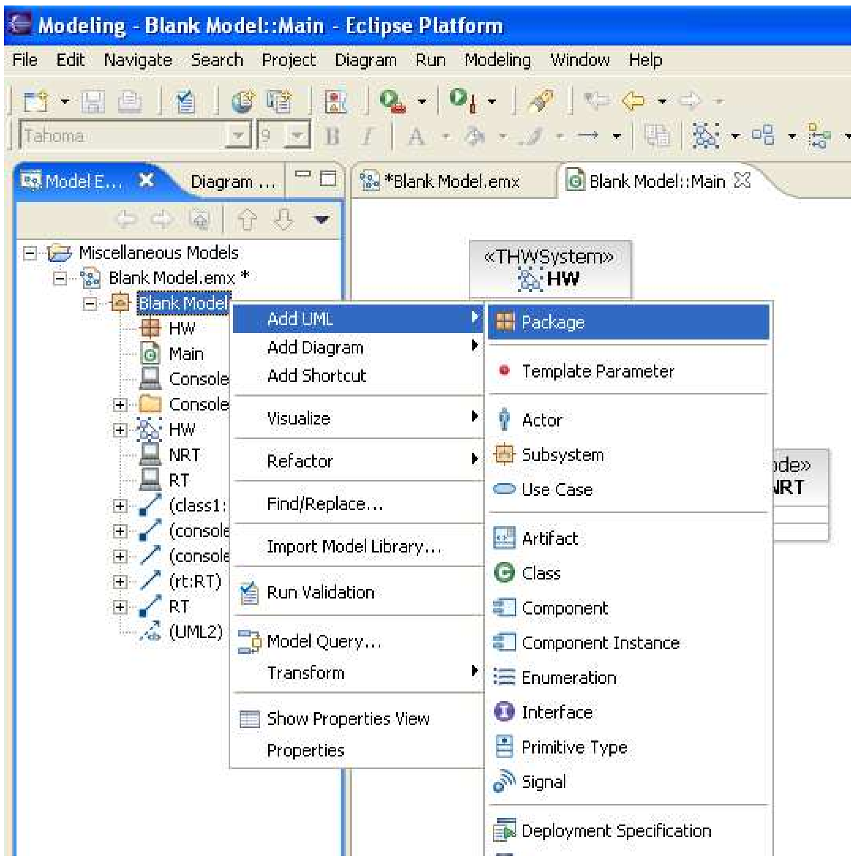


**Figure B.7** — Model elements can be grouped into packages.

## B.1.4 UML Views

Views on the model can be created by means of extra Class Diagrams. The creation of a new Class Diagram is shown in Figure B.8. Existing model elements can be dragged onto the new diagram, which thus provides a new view on those elements.

User actions: Create a new class diagram to contain the model of the software system in the SW package. This can be done by right-clicking on the package and choosing *Add Diagram* → *Class Diagram*. The freeform diagram created by default in the package can be deleted by right-clicking on it and selecting *Delete From Model* or simply pressing the *delete* button on the keyboard.

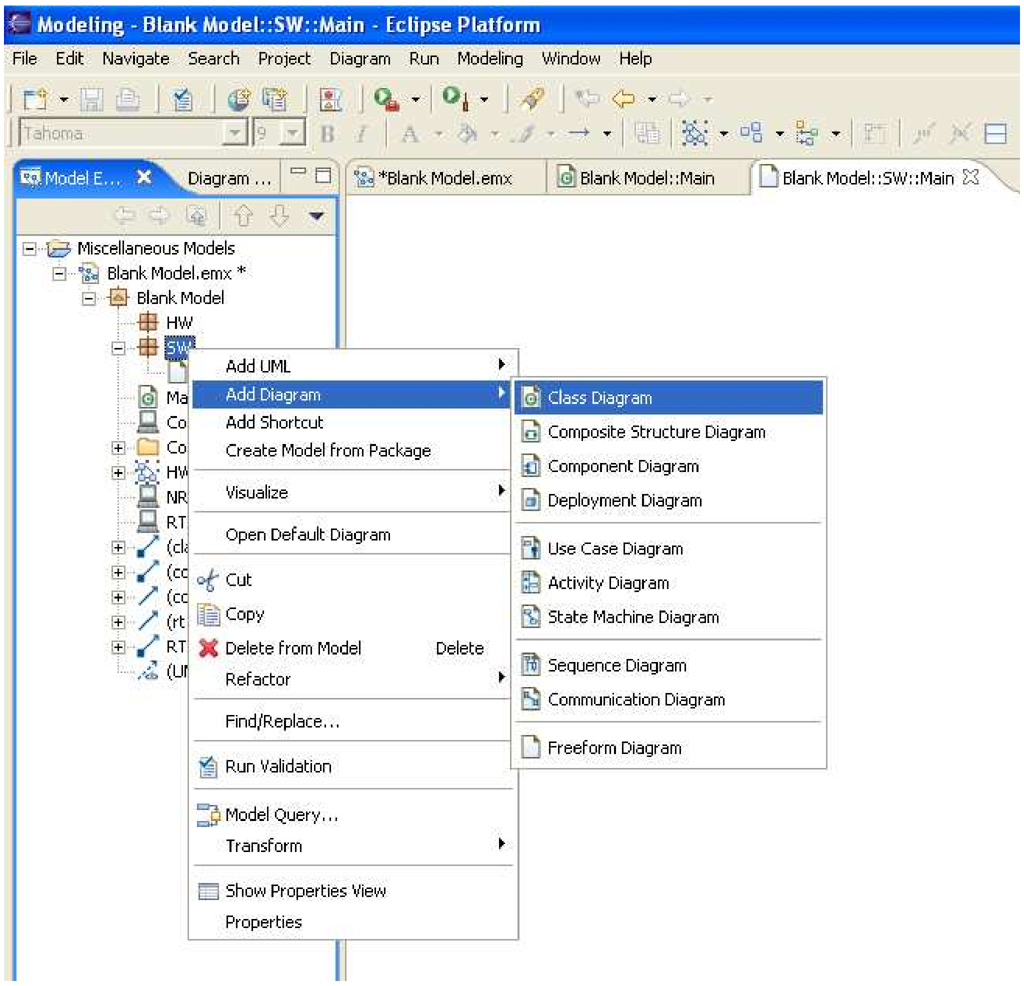


**Figure B.8** — Create a new class diagram in the software (SW) package.

User actions: Now create the software system on the new class diagram. Elements can be added in the same way as described for the hardware system. The result is shown in Figure B.9.

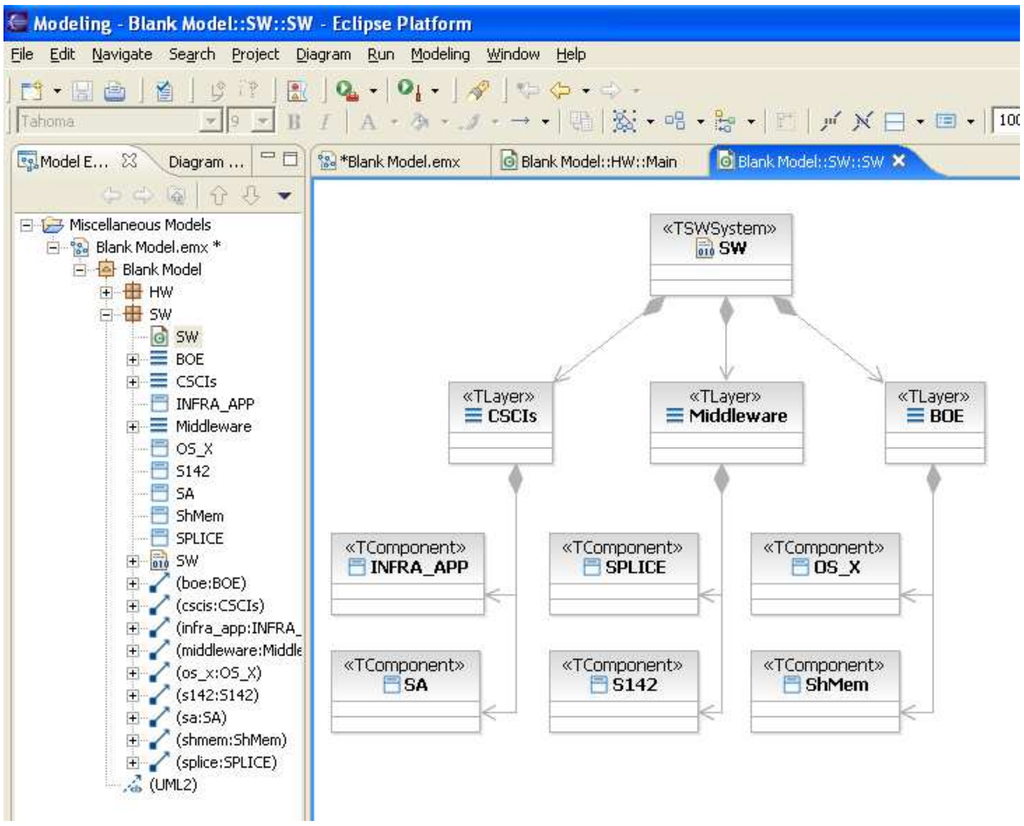


**Figure B.9 —** Create the software model inside the SW package.

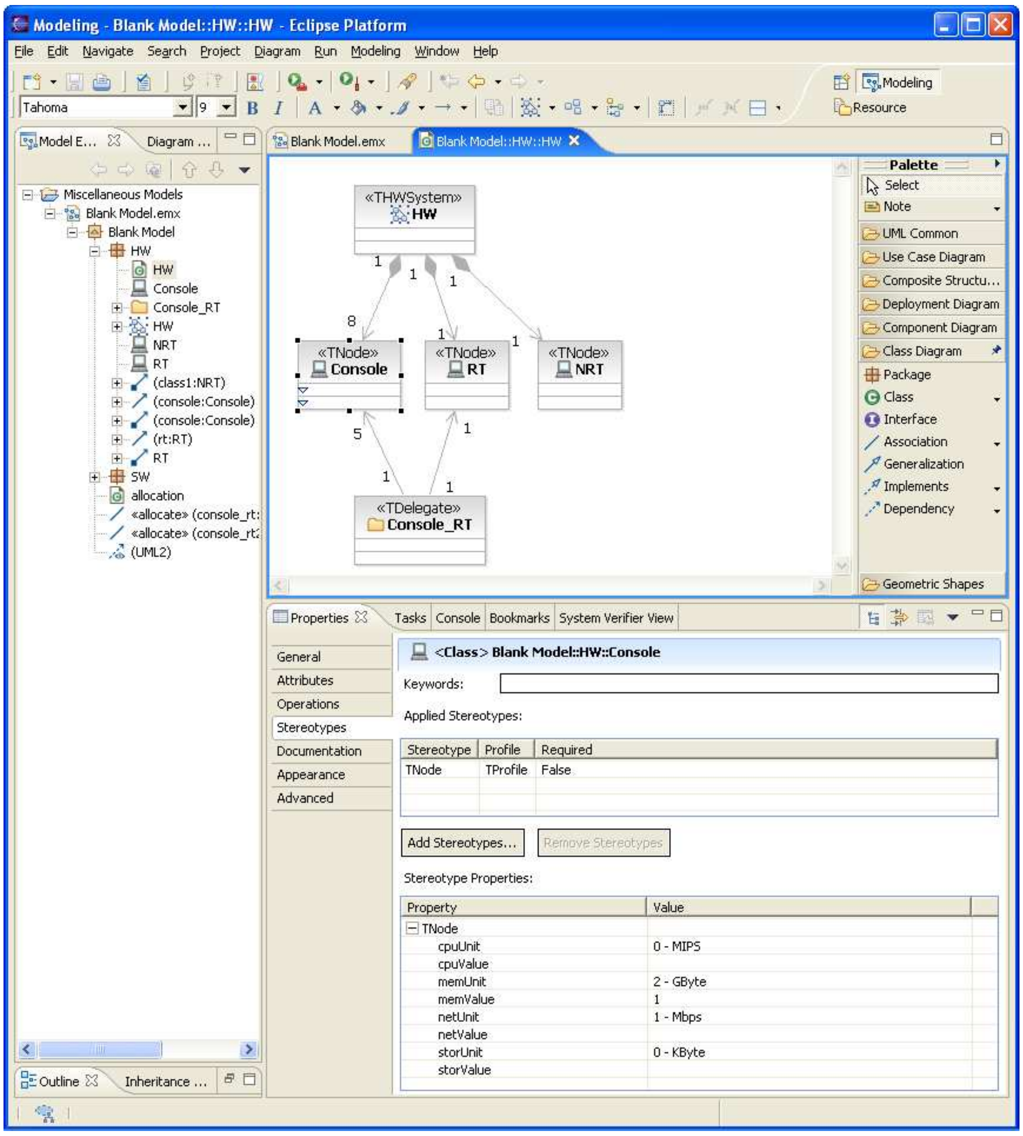
## B.2 Budgets

Several ways are available to define budgets. Stereotypes contain predefined budgets as tags and custom budgets can be added to a class as attributes.

### B.2.1 Tags

The budgets of CPU, memory, storage and networking are predefined in the stereotypes. Setting the value for the memory budget for example, can be done as shown in Figure B.10.

User actions: To set the value of a predefined budget (i.e. defined as a tag in the stereotype), first select the model element for which the budget has to be set (for example the Console in Figure B.10). Go to the properties view at the bottom of the screen and select the *Stereotypes* tab. Now fill in the required budget value. In this figure, the memory value has been set to 1 and the unit has been changed (by means of the drop-down list) to GByte.



<Class> Blank Model::HW::Console

Keywords:

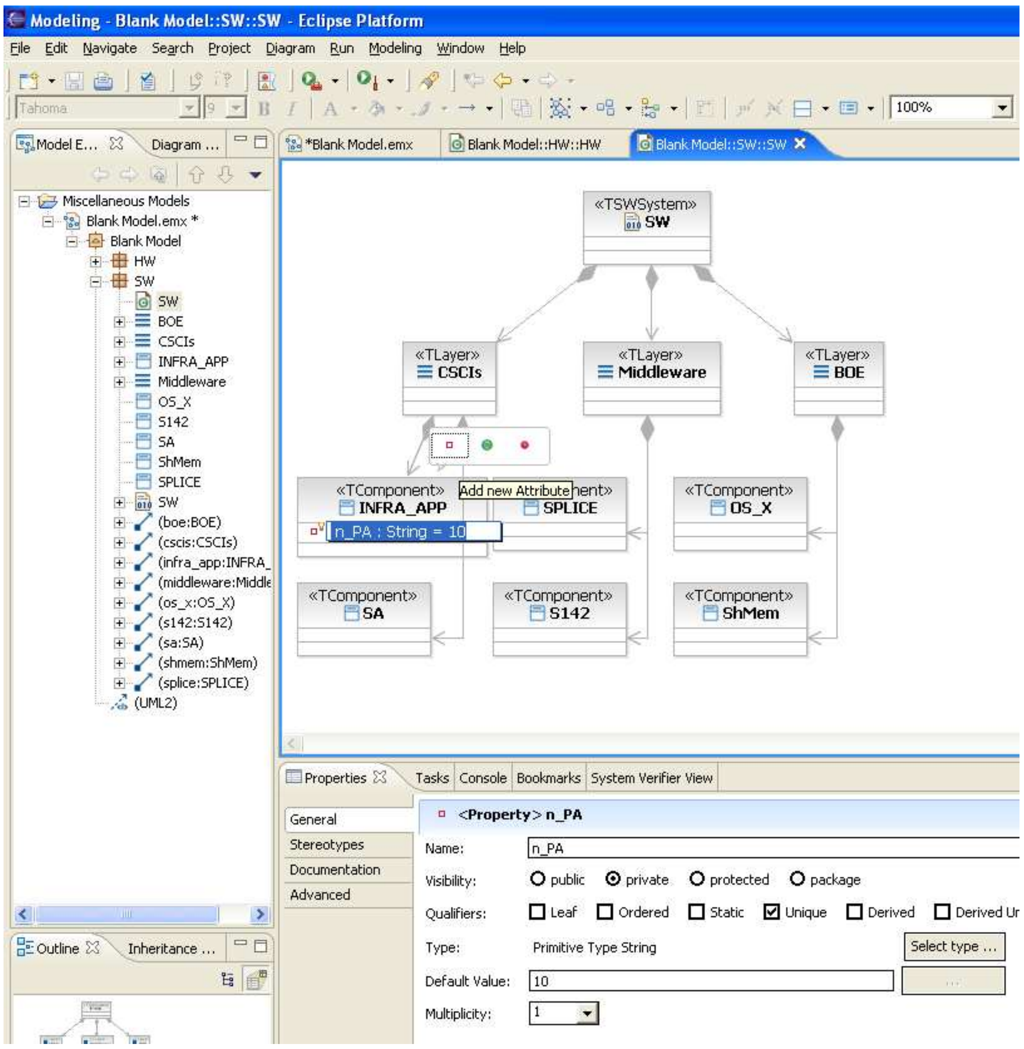
Applied Stereotypes:

| Stereotype | Profile | Required |
|---|---|---|
| TNode | TProfile | False |

Stereotype Properties:

| Property | Value |
|---|---|
| TNode | |
| cpuUnit | 0 - MIPS |
| cpuValue | |
| memUnit | 2 - GByte |
| memValue | 1 |
| netUnit | 1 - Mbps |
| netValue | |
| storUnit | 0 - KByte |
| storValue | |

**Figure B.10 —** Stereotype budget values



n_PA : String = 10

<Property> n_PA

Name: n_PA

Visibility: public [✓ private] protected package

Qualifiers: Leaf Ordered Static [✓ Unique] Derived Derived Ur

Type: Primitive Type String

Default Value: 10

Multiplicity: 1

**Figure B.11 —** Attribute budget values

### B.2.2 Attributes

Custom budgets can be added to model elements by means of attributes. The new budget of INFRA_APP in Figure B.11 has been given the name n_PA, the type String (all budgets have to be of type String, which can contain both numerical values and expressions) and the value 10.

User actions: Add a new attribute by hovering the mouse pointer over the model element and selecting the red square *Add new Attribute*.

The attribute consists of three parts: a name, type and value, which can be entered according to the following syntax: *name* : *type* = *value*. Names have to be unique and can only consist of alphanumerical characters and underscores. A name has to start with a letter.

User actions: Give the attribute the name *n_PA*, the type *String* and the value 10 by typing `n_PA : String = 10`. The properties view at the bottom can also be used to set these values.

### B.2.3 System Verifier View

Once budgets are created, their values can be inspected in the new System Verifier View. The view can be made visible as illustrated in Figure B.12.

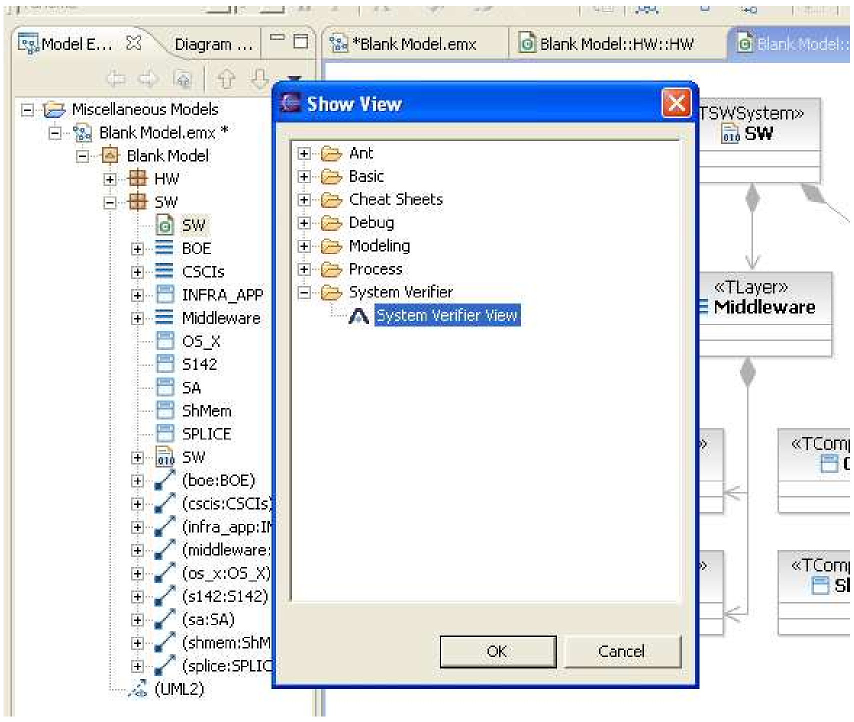
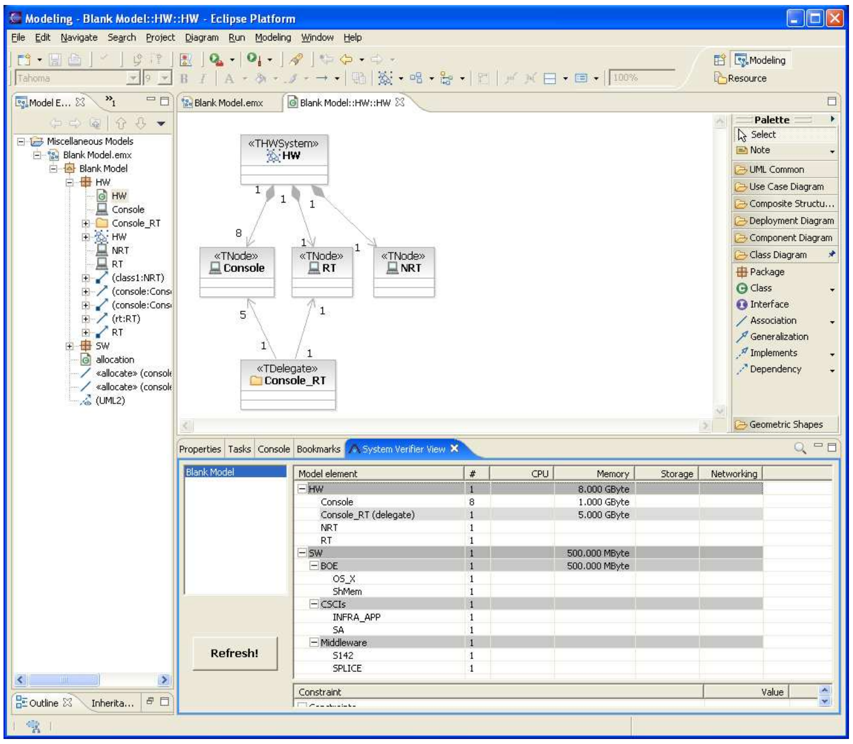


**Figure B.12 —** Show the System Verifier View

User actions: Select *Other...* from the *Window* → *Show View* menu. Then select the *System Verifier View* from the *System Verifier* category. The resulting view is shown in Figure B.13 and consists of four parts. Top left are the opened UML models. Top right is a table showing the structure of the model and budget values. Bottom right is a table showing constraints and bottom left is the button by which the entire view can be refreshed. The magnifying



| Model element | # | CPU | Memory | Storage | Networking |
|---|---|---|---|---|---|
| − HW | 1 | | 8.000 GByte | | |
| Console | 8 | | 1.000 GByte | | |
| Console_RT (delegate) | 1 | | 5.000 GByte | | |
| NRT | 1 | | | | |
| RT | 1 | | | | |
| − SW | 1 | | 500.000 MByte | | |
| − BOE | 1 | | 500.000 MByte | | |
| OS_X | 1 | | | | |
| ShMem | 1 | | | | |
| − CSCIs | 1 | | | | |
| INFRA_APP | 1 | | | | |
| SA | 1 | | | | |
| − Middleware | 1 | | | | |
| S142 | 1 | | | | |
| SPLICE | 1 | | | | |

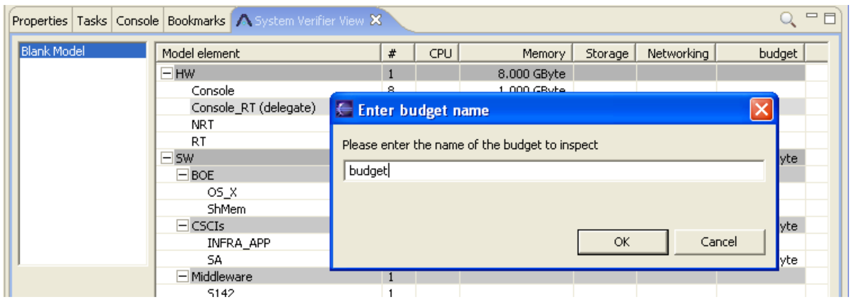
| Constraint | Value |
|---|---|

Modeling - Blank Model::HW::HW - Eclipse Platform
File Edit Navigate Search Project Diagram Run Modeling Window Help
Blank Model.emx
Blank Model::HW::HW
«THWSystem» HW
«TNode» Console
«TNode» RT
«TNode» NRT
«TDelegate» Console_RT
Palette
Properties Tasks Console Bookmarks System Verifier View
Blank Model
Refresh!

**Figure B.13 —** The System Verifier View.

glass at the top-right of the System Verifier View as shown in Figure B.14 and can be used to inspect the values of custom budgets, such as the *SA.budget*.

User actions: Click on the magnifying glass and type the name of the budget to inspect to include it in the view. Double clicking on the column header removes the column from the view. Rows are given certain colors to easily distinguish model elements of different types.



Enter budget name

Please enter the name of the budget to inspect

budget

Properties Tasks Console Bookmarks System Verifier View
Blank Model
Model element # CPU Memory Storage Networking budget
OK
Cancel

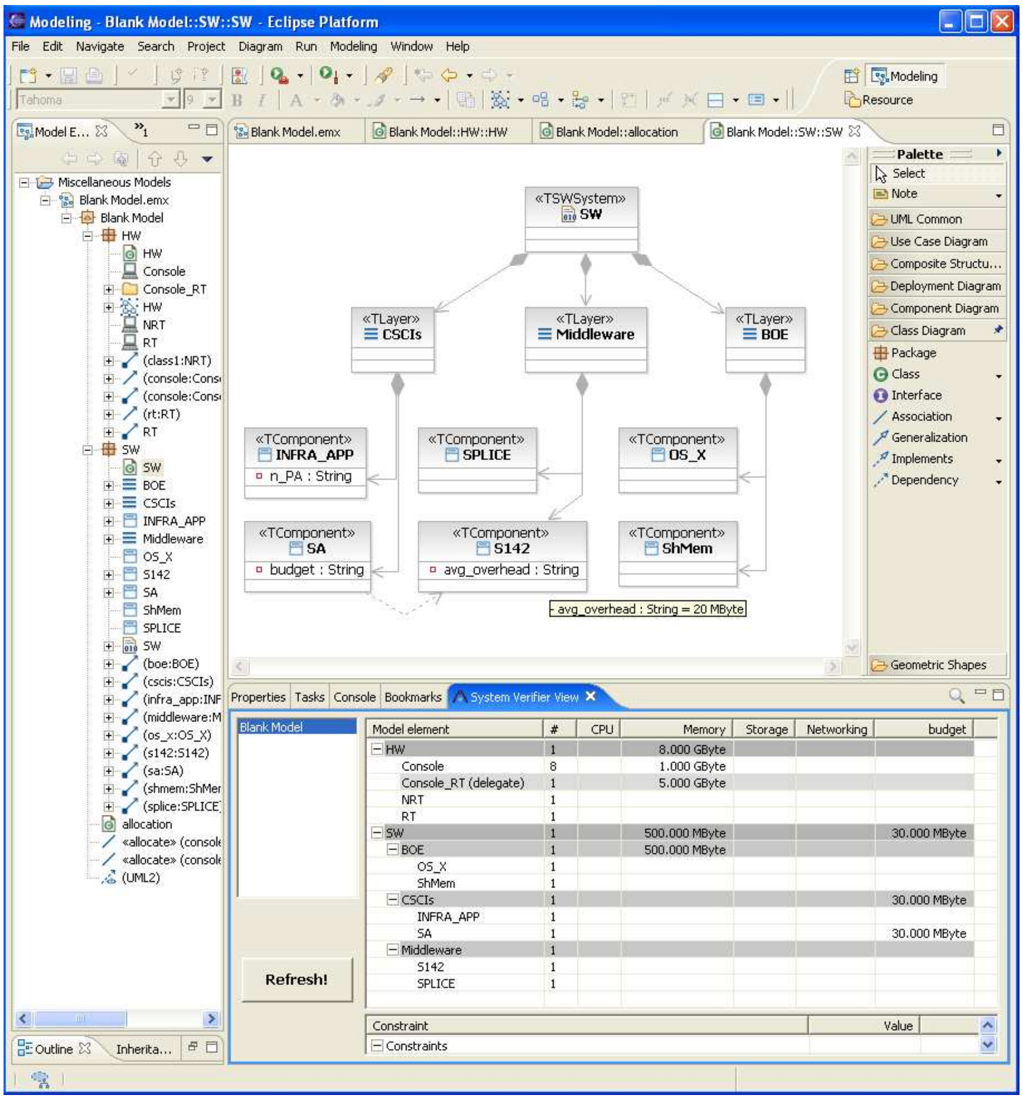
**Figure B.14 —** Show budget values in the System Verifier View

Depending on its level in the decomposition hierarchy, the row of an element is colored a certain shade of gray. When a budget has been defined for an element, but is smaller than the sum of the budgets of its children, the row is colored orange. This is done to warn the engineer that the defined value has been 'overwritten' by the larger sum of the children.

## B.2.4 Dependencies

Figure B.15 shows the addition of a dependency relationship between SA and S142.

User actions: To make budgets of SA dependent of S142 budgets, create a dependency relationship from SA to S142 in the same way as other relationships (for instance the Composition Association of Figure B.3). Then, to make a single budget dependent, change its value to refer to a budget of a dependency.



| Model element | # | CPU | Memory | Storage | Networking | budget |
|---|---|---|---|---|---|---|
| HW | 1 | | 8.000 GByte | | | |
| Console | 8 | | 1.000 GByte | | | |
| Console_RT (delegate) | 1 | | 5.000 GByte | | | |
| NRT | 1 | | | | | |
| RT | 1 | | | | | |
| SW | 1 | | 500.000 MByte | | | 30.000 MByte |
| BOE | 1 | | 500.000 MByte | | | |
| OS_X | 1 | | | | | |
| ShMem | 1 | | | | | |
| CSCIs | 1 | | | | | 30.000 MByte |
| INFRA_APP | 1 | | | | | |
| SA | 1 | | | | | 30.000 MByte |
| Middleware | 1 | | | | | |
| S142 | 1 | | | | | |
| SPLICE | 1 | | | | | |

| Constraint | Value |
|---|---|
| Constraints | |

**Figure B.15 —** New dependency

For example, the budget *SA.budget* can be made dependent on *S142.avg_overhead* by setting its value to *10+avg_overhead*. The SA.budget thus contains a reference to S142.avg_overhead and depends on the value of that budget. This is only allowed if a dependency relationship exists between SA and S142, as illustrated in Figure B.15. With S142.avg_overhead having a value of 20 MByte, the net value of SA.budget becomes 30 MByte. The value of a budget can be made visible by hovering the mouse pointer over the budget, as shown for the avg_overhead. The dependency of Figure B.15 can be used by both the SA.budget and S142.avg_overhead budgets, which have the following values:

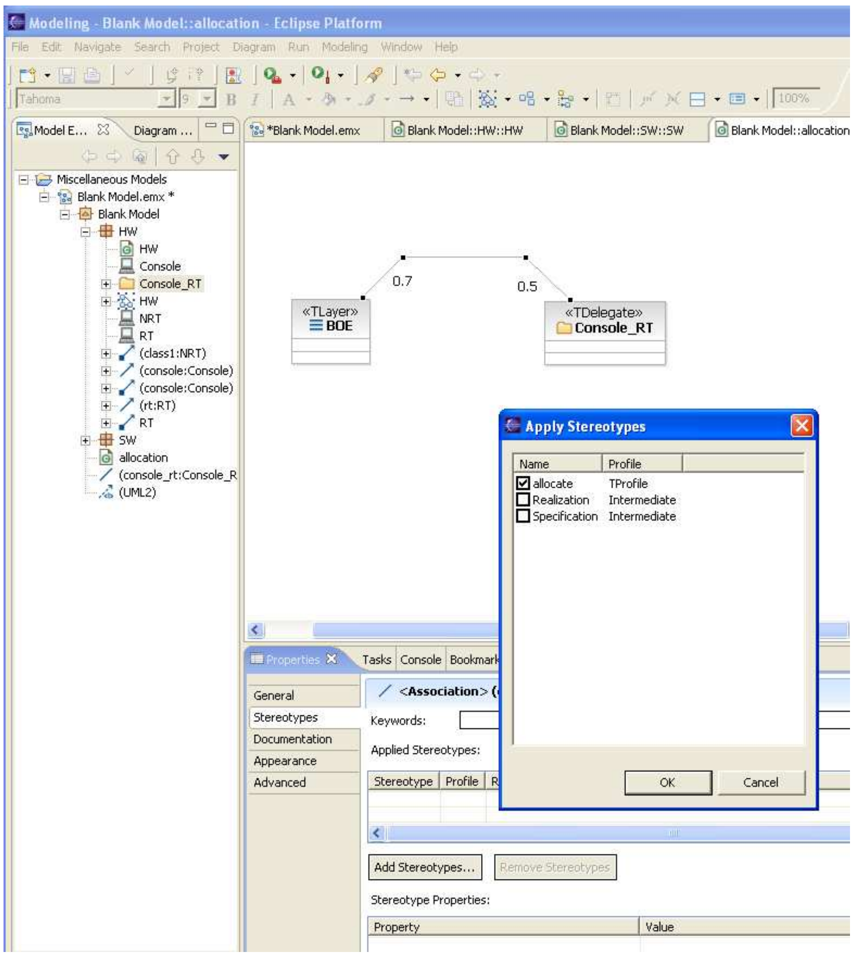
**SA.budget** 10 + avg_overhead

**S142.avg_overhead** 20 MByte

The dependency also makes the budgets of SA available to S142. A reference to *budget* from S142 will therefore have the value 20 MByte.

## B.3 Allocations

Software is related to hardware via allocation relationships. These are default UML associations with the «allocate» stereotype. Adding the «allocate» stereotype to an association is illustrated in Figure B.16.



| Name | Profile |
|---|---|
| ☑ allocate | TProfile |
| ☐ Realization | Intermediate |
| ☐ Specification | Intermediate |

**Figure B.16 —** New allocations

User actions: To allocate a software part to a hardware part, first create a new class diagram (a new view) to hold the allocation relationships (the class diagram is called *allocation* in the screenshot). Now drag the BOE layer from the SW package onto the diagram area (do not create a new BOE!) and do the same for the Console_RT delegate of the HW package. They then both show in the diagram, as illustrated in Figure B.16.

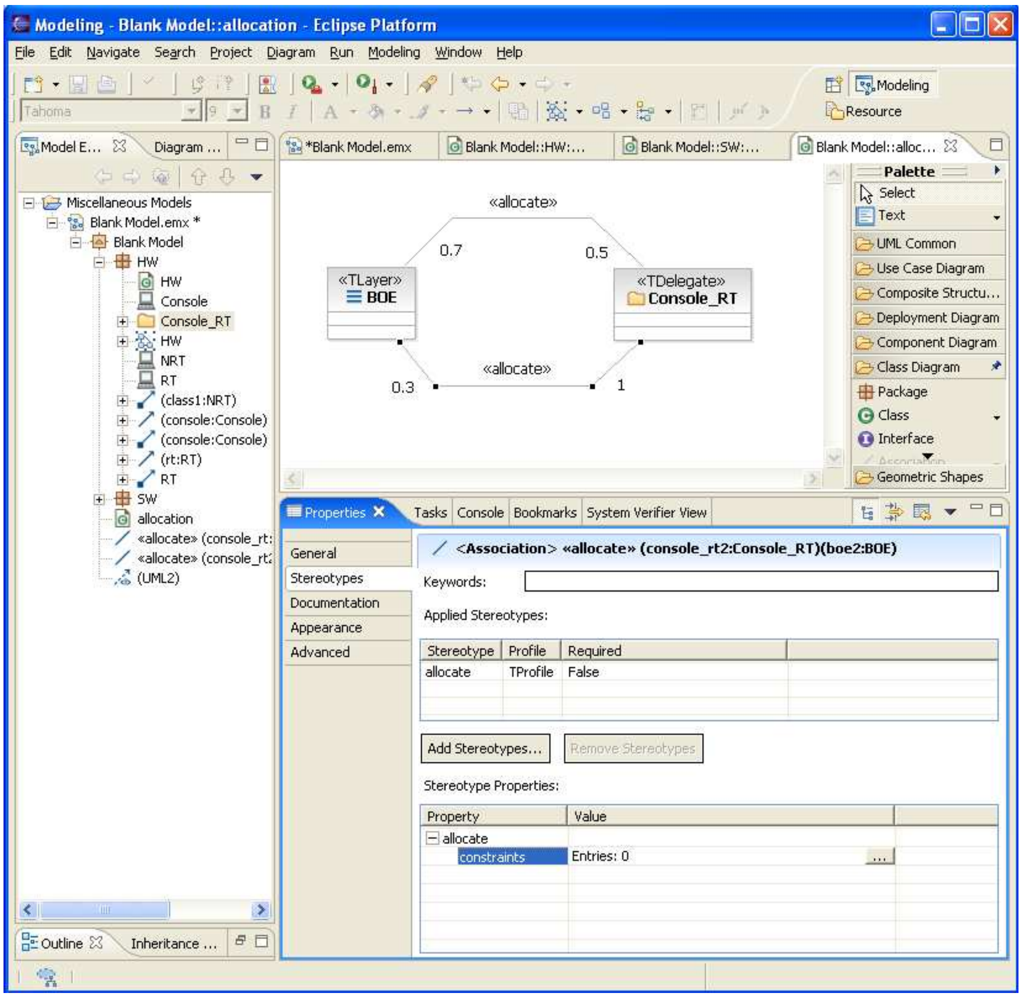
An allocation can be added by creating an association relationship between the elements and stereotyping it. First create an association relationship between the BOE and the Console_RT. Then add the stereotype by selecting the association and choosing *Add Stereotypes* in the *Stereotypes* tab of the *Properties* view. Here, the «allocate» stereotype is added.

The multiplicities at the allocation relationships represent the ratios of the allocation. The example of Figure B.16 shows that already 70% (0.7) of the BOE is allocated to 50% (0.5) of the Console_RT delegate. Create another allocation relationship which allocates 30% (0.3) of the BOE to 100% (1) of the Console_RT. The BOE is then completely allocated.

User actions: Change the default multiplicities of the allocation relationships to the values as shown in Figure B.16.

### B.3.1 Constraints

Constraints are to be added to the *constraints* tag of the «allocate» stereotype. This is illustrated in Figures B.17 and B.18.



<Association> «allocate» (console_rt2:Console_RT)(boe2:BOE)

Keywords:

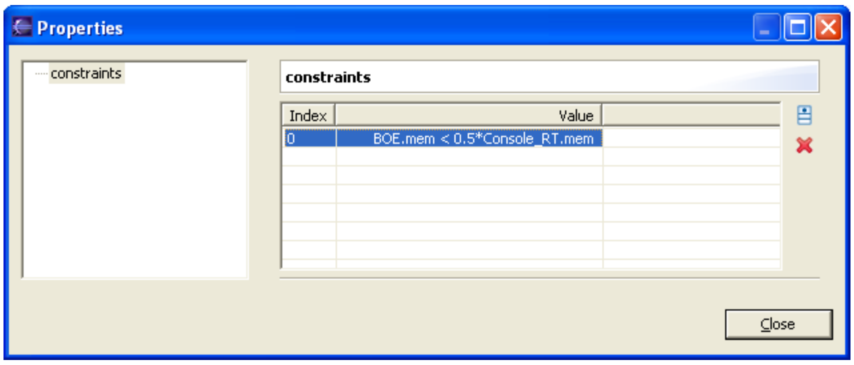
Applied Stereotypes:

| Stereotype | Profile | Required |
|---|---|---|
| allocate | TProfile | False |

Stereotype Properties:

| Property | Value |
|---|---|
| allocate | |
| constraints | Entries: 0 |

Figure B.17 — Create new constraint (1/2)



| Index | Value |
|---|---|
| 0 | BOE.mem < 0.5*Console_RT.mem |

**Figure B.18 —** Create new constraint (2/2)

User actions: To add constraints, select the corresponding allocation relationship and select the *Stereotypes* tab of the properties view. Subsequently select the '...' button of the *constraints* tag, as shown in Figure B.17. This results in the screen of Figure B.18.

User actions: Specify a constraint by pressing the *Insert New String* button (above the red cross in Figure B.18) and defining the expression representing the constraint. In this case `BOE.mem < 0.5*Console_RT.mem`.

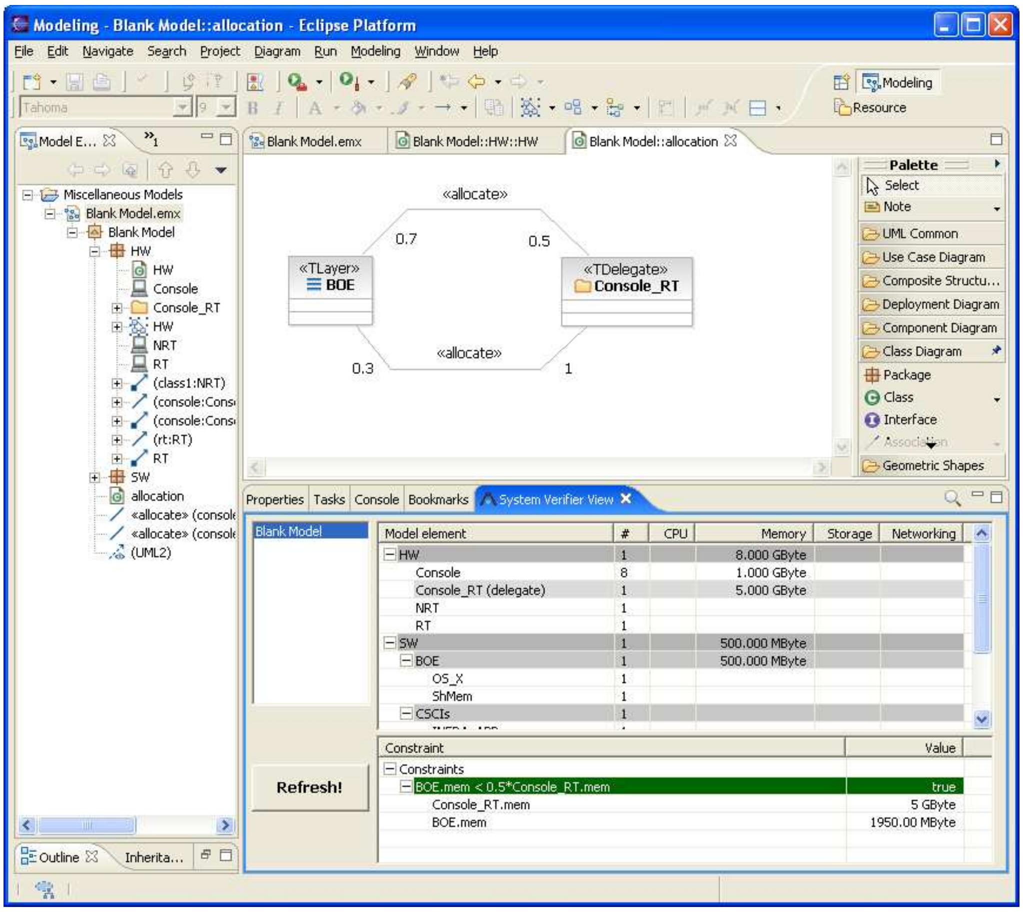
More constraints can be added by pressing the *Insert New String* button repeatedly. The constraint specified in Figure B.18 states that the memory budget of BOE has to be less than half the memory budget of Console_RT.

As many constraints can be added as required.

## B.4 Verification

User actions: Click the *Refresh* button on the System Verifier View to update it with the latest information and constraints to be evaluated.

The bottom right part of the view shows the newly created constraint and the values of referenced budgets. The constraint is shown in green because the result of the evaluation is true. If the result would have been false, the constraint would have been colored red. Values of budgets referenced by constraints are shown below them.



| Model element | # | CPU | Memory | Storage | Networking |
|---|---|---|---|---|---|
| HW | 1 | | 8.000 GByte | | |
| Console | 8 | | 1.000 GByte | | |
| Console_RT (delegate) | 1 | | 5.000 GByte | | |
| NRT | 1 | | | | |
| RT | 1 | | | | |
| SW | 1 | | 500.000 MByte | | |
| BOE | 1 | | 500.000 MByte | | |
| OS_X | 1 | | | | |
| ShMem | 1 | | | | |
| CSCIs | 1 | | | | |

| Constraint | Value |
|---|---|
| Constraints | |
| BOE.mem < 0.5*Console_RT.mem | true |
| Console_RT.mem | 5 GByte |
| BOE.mem | 1950.00 MByte |

**Figure B.19 —** View constraint evaluation results