UNIVERSITY OF OSLO
Department of Informatics

MARTES 2006 at MoDELS 2006

International workshop on Modeling and Analysis of Real-Time and Embedded Systems

Research Report 343

S. Gerard, S. Graf, Ø. Haugen, I. Ober, B. Selic

ISBN 82-7368-299-4
ISSN 0806-3036

October 2006

MARTES 2006 at MoDELS 2006 in Genoa, Italy

09:30 - 09:40	Introduction *The organizers*	

Session I : Visions and standards (1h20)

09:40 - 10:30	Keynote talk: to be announced	
10:30 - 11:00	Towards a UML-based Modeling Standard for Schedulability Analysis of Real-time systems *H. Espinoza, J. Medina, H. Dubois, S. Gerard, F. Terrier*	5

Session II : Validation and tools (1h30)

11:30 - 12:00	A3S method and tools for analysis of real time embedded systems *S. Rouxel, G. Gogniat, J-P. Diguet, J-L. Philippe, C. Moy*	15
12:00 - 12:20	Modeling with logical time in UML for real-time embedded system design *Ch. André, A. Cuccuru, R. de Simone, Th. Gautier, F. Mallet, and JP. Talpin* (work-in-progress)	27
12:20 - 12:40	Analysis and Modeling of Real-Time Systems with Mechatronic UML taking Clock Drift into Account *H. Giese, S. Henkler, and M. Hirsch* (work-in-progress)	41
12:40 - 13:00	Analyzing Robustness of UML State Machines *S. Prochnow, G. Schaefer, K. Bell, and R. von Hanxleden* (work-in-progress)	61

Session III : New language ideas (1h30)

14:30 - 15:00	Time Exceptions in Sequence Diagrams *O. Halvorsen, R. K. Runde, Ø. Haugen*	81
15:00 - 15:30	An Approach to Performance Modeling of Software Product Lines *J. A. Street and H. Gomaa*	101
15:30 - 16:00	Concurrency and Real time specifications in UML *K. Lano, K. Androutsopolous, D. Clark*	117

Session IV : Experience reports (1h45)

16:30 - 16:50	Modeling WS-BPEL with RT-UML Diagrams *M.-E. Cambronero, J-J. Pardo, G. Diaz, and V. Valero* (work-in-progress)	137
16:50 - 17:10	Applying Model Intelligence Frameworks for Deployment Problem in Real Time and Embedded Systems *A. Nechypurenko, E. Wuchner, J. White, D. C. Schmidt* (position paper)	151
17:10 - 17:40	An Experience in modeling real-time systems with SysML *P. Colombo, V. Del Bianco, L. Lavazza, A. Coen-Porisini* (work-in-progress)	157
17:40 – 18:15	Discussion	

Towards a UML-Based Modeling Standard for Schedulability Analysis of Real-Time Systems

Huáscar Espinoza, Julio Medina*, Hubert Dubois, Sébastien Gérard, and François Terrier
CEA Saclay, DRT/DTSI/SOL/L-LSP, F-91191, Gif-sur-Yvette Cedex, France
{huascar.espinoza, julio.medina, hubert.dubois, sebastien.gerard}@cea.fr

Abstract

This paper presents a generic modeling framework for specifying analyzable UML models of real-time systems. The underlying work was carried out in the context of the OMG initiative for standardizing a UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), which replaces and extends the profile for Schedulability, Performance, and Time Specification (SPT). We focus on some usability and flexibility weaknesses of the SPT modeling framework and describe the improvements brought off in MARTE. The UML extensions include new analysis-specific concepts aimed to support a broader range of quantitative analysis techniques. Additionally, the modeling approach involves a precise language to unambiguously declare and annotate non-functional properties with extended data types, variables, and complex expressions. As illustration, we show how this profile is used in the ACCORD|$_{UML}$ methodology to enable schedulability analysis of annotated UML models.

1. Introduction

The creation of models suitable for predictive and quantitative analysis is a key concern in any methodology for real-time systems development. The tighter the link between the real-time annotations and the modeling artifacts used along the development process, the sooner the verifications for detecting and removing errors can be done, and the easier the design tradeoffs for resources allocation become [11]. Indeed, the ability to evaluate Non-Functional Properties (NFPs) based on mathematically derived results stemming from accurate models is far better than relying exclusively on intuition. The real-time systems community has invested special efforts in incorporating the abilities to describe timeliness properties with enough expressive power, while still preserving the modeling abstraction level used by practitioners.

Like in other software engineering domains, the necessity to raise the level of abstraction, in order to increase designers' productivity and cope with complexity, as well as the facility that it brings to communicate design intents and generate documentation, have encouraged the usage of the Unified Modeling Language (UML) [13] in the real-time domain. As a standard, it adds a number of other well-known advantages that make it a primary modeling language along the software life cycle. Moreover, the Model Driven Development paradigm, and particularly the Model Driven Architecture (MDA) initiative [12] whose modeling support is provided by UML, lead a promising approach for reducing time-to-market and enrich software engineering practices, by moving the development process from lines-of-code to coarser-grained architectural elements.

However, UML per se is somehow still imprecise, and therefore in many cases insufficient for enabling quantitative analysis of real-time systems [2], [3], [14]. The question of how to add real-time modeling capabilities to UML has been addressed in different research and industrial contexts, mostly by the definition of "profiles", which is the built-in mechanism to extend UML to different specialized domains. Thus, for example, the OMEGA [14] profile provides a precise formal semantics dedicated to temporal aspects, UML-MAST [10] considers timing properties and schedulability analysis for distributed real-time systems, and recently a (draft) UML profile for the Avionics Architecture Description Language (AADL) has been proposed [19], which would allow to easily integrate AADL analysis tools with UML CASE tools. Profiles are a powerful extension mechanism; however, most of these profiles suffer a lack of standardization. This is very problematic as the idea of a standard language would get lost.

Standardization provides common semantics and notations, while adopting the best practices for modeling real-time systems. Standards are also a cost effective solution, since interoperability allows using different tools on the same model, and because of the lower overall training costs. Additionally, this helps to manage the risk on the evolution of the tools providers market, reducing the dependency on one single tool. As an industrial standardization board, the Object Management Group (OMG) has been carrying out important efforts to provide

[*] Post-Doctoral internship. Grupo de Computadores y Tiempo Real, Universidad de Cantabria, Spain.

UML extensions for modeling temporal properties as well as other non-functional aspects of computer systems.

The UML Profile for "Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" (QoS&FT) [5] makes up part of these efforts. It defines a framework to define and annotate QoS properties in UML models (QoS is here used as a general term to denote NFPs). The QoS&FT profile provides a flexible mechanism to define most common QoS characteristics for different application domains, by means of QoS catalogs. The QoS&FT annotation mechanism is supported on a two-step process which implies the creation of extra objects required just for annotation purposes. This two-step process, however, requires too much effort for the users and may induce not readable models [17]. Moreover, QoS&FT ignores some necessary aspects to model NFP specifications in the real-time domain, such as variables specification, and complex expressions (e.g., time expressions).

The recent SysML language [8], is being considered to be adopted by the OMG as a language to model complex systems, including software, hardware, facilities, and process aspects. Although still insufficient for the real-time domain, SysML provides some useful mechanisms to model this kind of concerns. For instance, the "block" modular unit provides capabilities to specify reusable "model-oriented components", defining a collection of features that includes parameters, constraints, and value specifications qualified by measurement units.

A third OMG standard intended to support non-functional annotations is the UML profile for Schedulability, Performance and Time specification (SPT) [1]. It provided a straightforward annotation mechanism with a minimal set of common annotations to perform very basic quantitative analysis (essentially RMA and queuing theory based analysis). Nevertheless, its structure was not flexible enough to allow for new user-defined annotations or for different analysis techniques. Most experiences of the profile have led to a number of significant suggestions for improvement and consolidation [2, 3] deemed too disruptive for a simple revision. This included also new requirements for specifying both, software, and hardware aspects, MDA compliance for defining separately the platform, the application and allocation models, and modeling of other kinds of NFPs such as power consumption or memory size.

Hence, the OMG called for the development of a new UML profile for the Modeling and Analysis of Real-Time and Embedded systems (MARTE) [4]. The results presented in this paper were performed in the framework of the ProMARTE [20] working team, and comprise some of our proposals integrated in the definition of this new profile [18]. We particularly deal with MARTE modeling capabilities to enable predictive quantitative analyses, namely schedulability. We incorporate in MARTE a number of additional features while reduces the implicit complexity of QoS&FT. MARTE also introduces a precise value specification language that supports symbolic variables and complex expressions for specifying NFPs. But mainly, this framework provides a common modeling framework for different schedulability analysis techniques by factorizing out concepts used in different analysis models.

The paper is organized as follows. In the next section, we present a quick view of the MARTE profile. Section 3 describes its modeling framework to specify NFPs. Section 4 describes the framework for Schedulability Analysis Modeling (SAM). In Section 5, we show the usage of the profile by means of an example in the framework of the ACCORD$|_{\text{UML}}$ methodology [7].

2. The UML profile for MARTE

UML uses the metamodeling technique to define its abstract syntax. Metamodels define languages enabling to express models. This means that a metamodel describes the various kinds of contained model elements and the way they are arranged, related and constrained. Hence, a UML model is said to *conform* to its metamodel.

Profiles customize UML for a specific domain or purpose using extension mechanisms able to enrich the semantics and notation of the metamodel elements. A *stereotype* is the basic notion that allows extending UML. A stereotype can be viewed as a subclass of an existing UML concept, which provides the capability of modeling domain-specific concepts or patterns. Stereotypes may have typed properties called *tag definitions*, which may represent attributes or relations with other metamodel elements. Complementary, stereotypes can be also influenced by restrictions expressed in *constraints*.

The MARTE standard comprises both a conceptual *domain model* and the concrete *UML extension profile*. The first one defines key concepts and relationships between them used for describing real-time computing systems. The second one is the specification of how the elements of the domain model are realized by means of stereotypes, and how they extend UML metaclasses.

As Figure 1 shows, MARTE is structured around a core package (TCRM), which describes modeling constructs for time, resources, concurrency, allocation and NFPs, and two main groups of specialized packages that use and refine the core one. One group to model development features of real-time and embedded systems (RTEM), and the other one to annotate real-time models so as to support quantitative analysis (GQAM).

RTEM provides means to define application-modeling patterns, and refined modeling constructs to model hardware and software execution platforms.

On the other hand, the GQAM package provides a generic basis for different quantitative analysis sub-domains. The GQAM package supports two main sub-profiles for *schedulability analysis* to predict whether a set of software tasks meets its timing constraints, and *performance analysis* to determine if a system with non-deterministic behavior can provide adequate performance. Additionally, the profile structure allows for adding further analysis domains, such as power consumption, memory use or reliability. It is the intention to encourage modular sub-profiles for such domains.

Figure 1 also describes some of the main potential actors that may use this specification. Thus, *model designers* are dedicated to define the hardware and software architecture of real-time systems. *Model analysts* are modelers concerned with annotating system models in order to perform specific analysis techniques. Execution *platform providers* are developers and vendors of run-time technologies (hardware- or/and software-based platforms) such as Real-Time CORBA, real-time operating systems and specific hardware components.

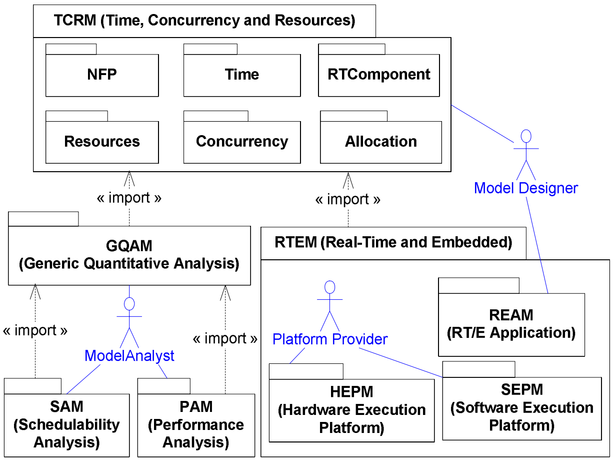


Figure 1. Architecture of the MARTE Profile

In the context of UML-based quantitative analysis, annotating NFPs is of fundamental relevance and implies a number of modeling mechanisms for complexity management in the development cycle, but mainly, for specifying time expressions semantically well formed. The following section describes the NFP modeling package as the basic framework to model quantitative analysis aspects.

3. The NFP Modeling Framework

The model of a computing system describes its architecture and behavior by means of model elements (e.g., resources, resources services, behavior features, logical operations, configurations modes, modeling views), and the properties of those model elements. It is common to group element properties into two categories: functional properties, which are primarily concerned with the purpose of an application (i.e., what it does at run time); and non-functional properties (NFPs), which are more concerned with its fitness for purpose (i.e., how well it does it or it has to do it) [18]. NFPs provides information about different characteristics, as for example throughput, delays, overheads, scheduling policies, correctness, security, memory usage, and so on.

The NFP modeling framework [17][18] is especially focused on formalizing a set of modeling UML constructs to specify non-functional information in a precise way. In fact, one of the common criticisms to the SPT profile is the very superficial semantic information that it provides; in particular, for promoting common understanding of specification and exchange of specifications between different tools. Thus, different tools or analysis techniques could use different metrics for the same concept. For instance, a task *priority* attribute, from a scheduling perspective, is defined in the context of a "priority" scale with either increasing or alternatively decreasing numerical values. Unless we properly qualify NFPs, no common meaning will be adopted, regardless of how detailed and expressive a property name might be. The NFP modeling framework attempts to overcome these kinds of ambiguities.

Figure 2 shows a partial view of the metamodel that supports this framework. This metamodel provides key constructs for modeling this kind of properties at two fundamental stages: declaration and specification. NFP declaration is intended to qualify and assign extended data types to NFP values. NFP specification allows describing values as constants, variables, complex expressions (including time expressions) in a standard textual language.

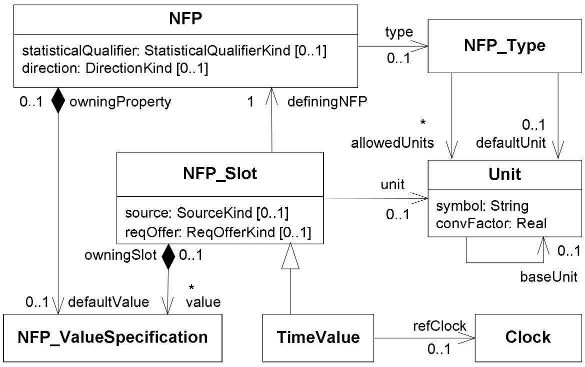


Figure 2. Partial view of the metamodel for NFP

NFPs are qualified by two basic attributes and by a data type (*NFP_Type*). Both attributes of NFPs, *statistical qualifier* and *direction*, have been adopted from the UML profile for QoS&FT [5]. A statistical qualifier indicates the type of "statistical" measure of a given property (e.g., maximum, minimum, range, mean, variance, percentile, distribution). The *direction* attribute (i.e., increasing or decreasing) defines the type of quality order relation in the allowed value domain of NFPs. Indeed, this allows multiple instances of NFP values to be compared with the relation "higher-quality-than" in order to identify what value represents the higher quality or importance. On the other hand, NFP Types add the ability to carry a measurement *unit* for NFP values associated with physical dimension measures. Additionally, we provide the capability of defining new user-specific units in terms of existing base units, through a given conversion factor.

Since a UML viewpoint, NFPs are implemented in MARTE as *tag definitions* qualified and typed as NFPs. MARTE defines a set of NFP Types commonly used in the real-time and embedded system domain. Examples of NFP Types are *Duration*, *Data Transmission Rate*, *Data Size*, *Power*, between others. As illustration, Figure 3 shows the declaration of the Duration NFP type.

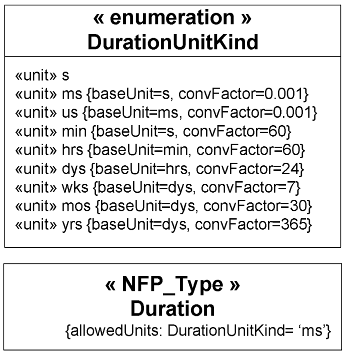

« enumeration »
DurationUnitKind
«unit» s
«unit» ms {baseUnit=s, convFactor=0.001}
«unit» us {baseUnit=ms, convFactor=0.001}
«unit» min {baseUnit=s, convFactor=60}
«unit» hrs {baseUnit=min, convFactor=60}
«unit» dys {baseUnit=hrs, convFactor=24}
«unit» wks {baseUnit=dys, convFactor=7}
«unit» mos {baseUnit=dys, convFactor=30}
«unit» yrs {baseUnit=dys, convFactor=365}
« NFP_Type »
Duration
{allowedUnits: DurationUnitKind= 'ms'}

Figure 3. Example of declaration of the Duration NFP type

A model of a system (which is considered to be expressed in UML) can be annotated by specific NFP values, expressing concepts from a given modeling concern or domain viewpoint (for example Schedulability Analysis). An annotated model element describes certain of its non-functional aspects by means of NFP annotations. These annotations are specified by the designer in the models and attached to different model elements. Examples are the *response time* of a *task* when executed, the *utilization* of a *resource*. Thus, a *NFP Slot* specifies that an annotated model element has a value or values for a specific NFP. The values in a slot must conform to the defining NFP of the slot (in type, multiplicity, etc.). NFP Slots include the attributes *source* to qualify different value versions (as for example *required*, *estimated*, or *calculated*), an associated measurement unit (e.g., ms, KB/s, MB) that overwrites the default value declared in the NFP Type definition, and a qualifier for describing the required or offered nature of a NFP value (*reqOffer*).

On the other hand, *NFP Value Specification* defines the textual expressions associated with NFP slots. The NFP modeling framework provides an abstract syntax and a grammar for specifying the values of NFPs. Indeed, while NFPs values are often assumed to be simple values, there are certain cases where it may be necessary to express such values in a more complex way. For example, it may be required for one NFP value to be related in some way to another. This requires both a way of referencing the value of another property as well as the ability to use expressions, such as arithmetic or time expressions. Thus, each value can be specified as a constant, as a variable, as a complex expression value, a special duration or instant expression, or as an interval of value specifications.

Table 1 shows typical examples of the notation for the body of value specifications.

NFP Value Specification	Example
Real Number	`1.2E-3`
Variable	`$timeout`
Ordered Collection	`(1, 2, 5, 88)`
Interval	`[1..251]`
DateTime	`12/01/06 12:00:00`
Duration (between two events)	`($startEvent, $endEvent)`
Duration (number of clock ticks)	`5*t{refClock}`
Constraints	`$deadline < $timeout + 5.0`
Logical Expression	`$V1=(($clients<6)?(exp(6)))`

Table 1. Examples of NFP Value Specifications

In Section 5, we show some examples for specifying NFP values and attaching them to UML elements by means of tagged values.

4. Modeling for Schedulability Analysis

All analysis methods use a simplified/abstract view of the system to analyze, which focuses on those aspects that are relevant to the associated analysis technique. Thus, the analysis modeling concepts rarely map one-for-one to application-level modeling concepts. Particularly, in schedulability analysis, a key abstraction is the notion of a unit of concurrency, representing some execution entity that requires the scheduling services of the system platform. However, application models typically do not show the units of scheduling explicitly. Instead, they are implied by the presence of model elements such as active objects and asynchronous messages.

The SPT profile already included a basic collection of UML extensions required to define analysis modeling views, particularly for performance and schedulability analyses. However, it was actually stated that they were insufficient for a number of analysis techniques and for certain real-time computing implementations, like distributed systems, or hierarchical scheduling [3]. As a consequence, most UML-based analysis methods did not use/refine the SPT profile as planned, instead they created their own refined (non-standard) extensions [10][14].

To overcome these problems, the SAM framework described in this section defines a collection of extended modeling concepts, as well as a set of generic and extensible/replaceable NFPs for them, oriented to model real-time computing systems from a wider range of schedulability analysis techniques perspective. MARTE provides a minimal set of common annotations for model-based schedulability analysis. This minimal set furnishes enough information to perform common schedulability analyses. However, each vendor is encouraged to supply specialized NFP annotations that extend this set in order to perform model analysis that is more extensive.

4.1 Domain model for schedulability analysis

The Schedulability Analisys Modeling (SAM) domain model is organized as in SPT under the concept of a *Real-Time Situation*. From the predictive point of view schedulability analysis models are intrinsically instance-based. Hence, a real-time situation is still a kind of analysis context that represents a specific situation of the system, working in a particular mode and configuration, and with concrete computational resources. Nevertheless, high-level descriptor-based models can also be established using the RTEM sub-profiles, and then concrete analysis models may be instantiated for specific analyses.

A real-time situation collects the relevant quantitative information required to perform specific analysis. Starting with the real-time situation and its elements, a tool can follow the links of the model to extract the information that it needs to perform the model analysis. A real-time situation is described by separated models associated with three generic modeling concerns (Figure 4):

- *Workload Situation*: a constant load of end-to-end responses triggered by external (e.g., environmental events) or internal (e.g., a timer) stimuli.
- *Behavior Execution*: a description of the executed actions chain as response to the workload, including access to shared resources and their services.
- *Resources Platform*: a concrete architecture of hardware and software computational resources used.

This separation of modeling concerns permits to organize the domain models into comprehensible parts. Moreover, these concerns may act as a set of design sub-views at the user modeling level, thus allowing to reduce changing impact and to facilitate evolution and reusability. However, this is not a mandatory user-model organization. Users may create different views, if desired, with basis on the fundamental SAM concepts.

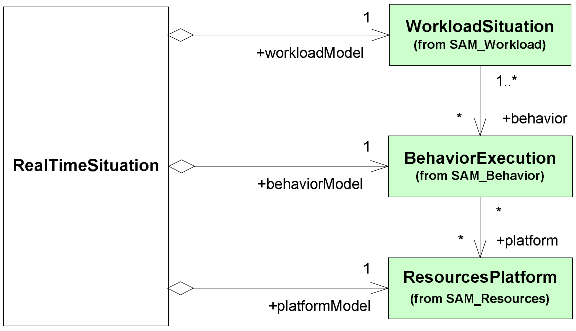


Figure 4. The modeling concerns of a Real Time Situation

4.1.1 The Workload domain model. This model describes the constructs required to specify the computation load on the system and the associated quantitative information about end-to-end stimuli, responses, and temporal requirements.

A *workload situation* of a real-time system is typically defined by the set of stimuli starting computations. In the SAM framework, we refer to an instance of a particular stimulus as an event occurrence. Since the stimulus can occur repeatedly, we refer to recurrence of events as a *trigger*.

A computation that is performed as a consequence of a trigger is referred to as the *response* to its event occurrences. Depending on the implementation nature of responses, they could be concretized in a single task executing in one processor or in dependent tasks into single or multiple processors. We do not include in this model the detailed behavior involving a response (we do it in the Behavior model).

Notice that the *trigger* and *response* concepts specify only end-to-end behaviors, specifically, their “cause” side and their “effect” side. Thus, in order to group these two concepts into a single modeling unit, we adopt the concept of *transaction*. A transaction [[15]] refers to the entire cause-effect end-to-end behavior describing a separate computation in the system. Hence, the set of particular transactions defining one load scenario for analysis purposes composes a single workload situation. A workload situation may correspond to a mode of system operation (e.g. starting mode, fault recovering, or normal operation) or a level of intensity of environment events.

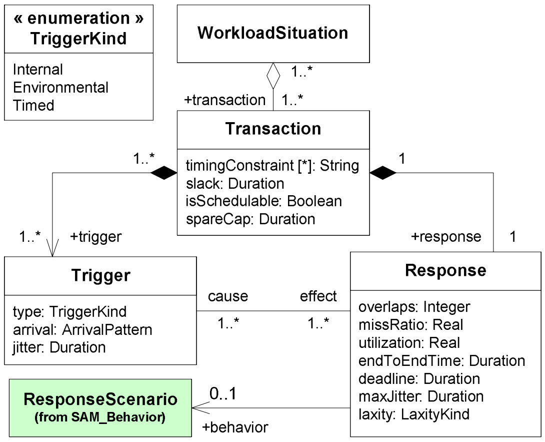


Figure 5. The Workload domain model

Additionally, the transaction, trigger and response concepts have a set of NFPs used for performing quantitative analysis. We define NFPs as tag definitions disambiguated by a statistical qualifier and a direction attribute (not shown here for space reasons). Different analysis techniques commonly use dissimilar NFPs. However, the SAM model defines a pre-declared set of NFPs useful for most of the analysis techniques.

For instance, a trigger is characterized by its *arrival pattern*, which is a structured data type containing concrete attributes, as for example *arrival kind*, *period*, *minimum arrival time*, *distribution function*, among others. On the other hand, responses specify a set of latency NFPs concretized by end-to-end delays or temporal requirements, e.g., *end-to-end time, deadlines*. Transactions may be also annotated with *efficiency* NFPs, as for example *slack* or *spare capacity* obtained from schedulability analysis tools. Timing constraints of transactions are expressed in the form of NFP constraints, which relates to observers (implemented by NFP variables) that must be attached at some point in the sequences of actions that describe the response scenario.

4.1.2 The Behavior domain model. Figure 6 summarizes the domain concepts for defining Behavior modeling aspects. In this model, the *behavior execution* concept serves to collect detailed descriptions of the behavior of responses, which are called in turn *response scenarios*. By response scenario we mean the specification of the smaller segment of code execution and their precedence and concurrence relationships. This modeling aspect is core for quantitative analysis in order to evaluate how the execution segments contend for use of the platform resources from a timing viewpoint.

In this manner, the detailed behavior of a given response is represented by an ordered series of step executions called scheduling actions *(SAction)*. Considering a holistic approach for the analysis, an RT action may represent the time it takes a piece of code execution as well as the sending of a message through a network. The ordering of SActions follows a predecessor-successor pattern, with the possibility of multiple concurrent successors and predecessors, stemming from concurrent thread joins and forks respectively. The granularity of a SAction is often a modeling choice that depends on the level of detail that is being considered. Hence, a SAction at one level of abstraction may be decomposed further into a set of finer-grained SActions. Response Scenarios use resource services for execution through the allocation of SActions to *schedulable entities*, and for other platform services, through calls to *shared resources* (*acquire* and *release* actions).

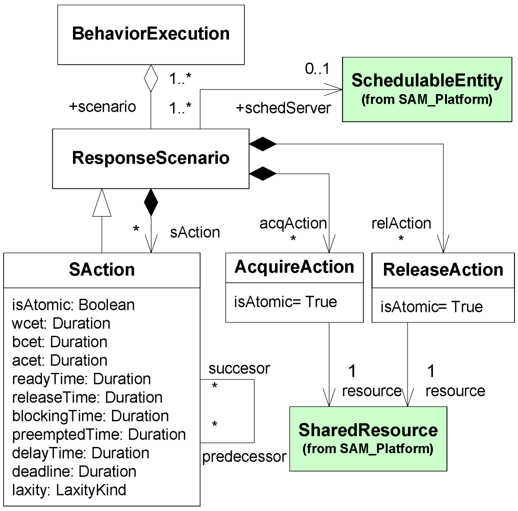


Figure 6. The Behavior domain model

The SAction concept is consistent with SPT, and hence it is characterized by a similar set of NFPs, enriched with some other and extensible latency properties like best and average case execution time.

4.1.3 The Platform domain model. In the SAM framework, the concept of *resources platform* matches to the model of engineering resources introduced in the SPT profile [[1]], [[6]]. This includes not only hardware resources (CPU, devices, backplane buses, network resources), but also software ones (threads, tasks, buffers). Figure 7 shows a framework to describe the platform of resources.

We shape an abstracted version of a more structured and detailed platform model (software and hardware execution MARTE platform sub-profiles), which is especially useful for expressing NFPs oriented to schedulability analysis and without distinguishing among different abstraction levels (hardware, RTOS or middleware).

The platform model consists of a set of resources with explicit NFPs. This model distinguishes two kinds of processing engines: *execution engines* (e.g., processors, coprocessors) and *communication engines* (e.g. networks, buses). For each, the SAM framework assigns generic NFPs. Specifically, *throughput* properties e.g., *processing rate* or *transmission rate*, *efficiency* properties e.g., *utilization*, *slack*, and *overhead* properties as for example *blocking times*, *interrupt overhead times*.

Schedulable entity is a kind of active protected resource that is used to execute SActions or complete Response Scenarios. In a RTOS this is the mechanism that represents a unit of concurrent execution, such as a task, a process, or a thread. In a network, it represents a channel or message management unit that can be characterized by concrete scheduling parameters (like the priority for a CAN bus).

Processing engines own *shared resources* as for example I/O devices, DMA channels, critical sections or network adapters. Shared resources are dynamically allocated to schedulable entities by means of an access policy. Common access policies are *FIFO*, *priority ceiling protocol*, *highest locker*, *priority queue*, and *priority inheritance protocol*.

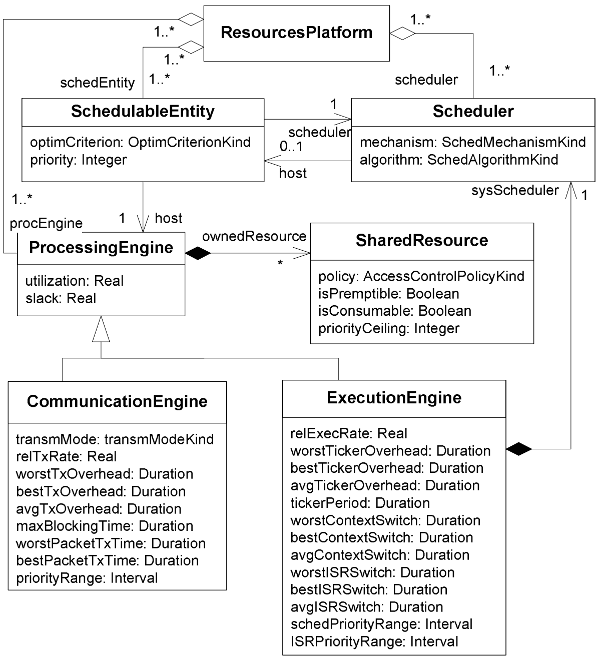


Figure 7. The Resources domain model

Schedulers can play two roles in this model, *system schedulers* (typically a RTOS scheduler) that offer the whole processing capacity of its associated base processors to its allocated schedulable entities, and other *secondary schedulers* that only provide the processing capacity offered by its hosting schedulable entity. This hierarchical structure is typically used in RTS when users are interested in applying dynamic scheduling on top of commercial RTOS supporting only static scheduling. Likewise, novel algorithms exist that make possible to perform real-time analysis of these hierarchical configurations of schedulers [16].

5. Using the SAM sub-profile

We now examine how the domain concepts previously presented can be represented (mapped) in the UML modeling space. The annotations have been made over a case study application for the real-time modeling and analysis of a simple distributed system for the teleoperated control of a robotized cell [10]. This example has been reformulated in the context of the ACCORD|$_{\text{UML}}$ methodology for developing real-time embedded systems. ACCORD|$_{\text{UML}}$ consists of a full MDA development process (and the underlying modeling and execution platforms) covering from requirements modeling, early quantitative analysis, to full implementation.

The application system is composed of two processors interconnected through a *CAN bus*. The first processor is a teleoperation station (*Station*); it hosts a GUI application, where the operator commands the robot and where information about the system status is displayed. The second processor (*Controller*) is an embedded microprocessor that implements the controller of the robot servos and its associated instrumentation.

The software architecture is described by means of the class diagram shown in Figure 8. The software of the Controller processor contains three active classes (called real time object –RTO in ACCORD|$_{\text{UML}}$) and a passive one which is used by the active classes to communicate. *Servo Controller* is a periodic RTO that is triggered by a ticker timer with a period of 5 ms. The *Reporter* RTO periodically acquires, and then notifies about, the status of the sensors. Its period is 100 ms. The *Command Manager* RTO is aperiodic and is activated by the arrival of a command message from the CAN bus.

The software of processor Station has the typical architecture of a GUI application. The Command Interpreter RTO handles the events that are generated by the operator using the GUI control elements. The Display Refresher RTO updates the GUI data by interpreting the status messages that it receives through the CAN bus. Display_Data is a protected object that provides the embodied data to the RTO in a safe way. Both processors have a specific communication software library and a background task for managing the communication protocol.

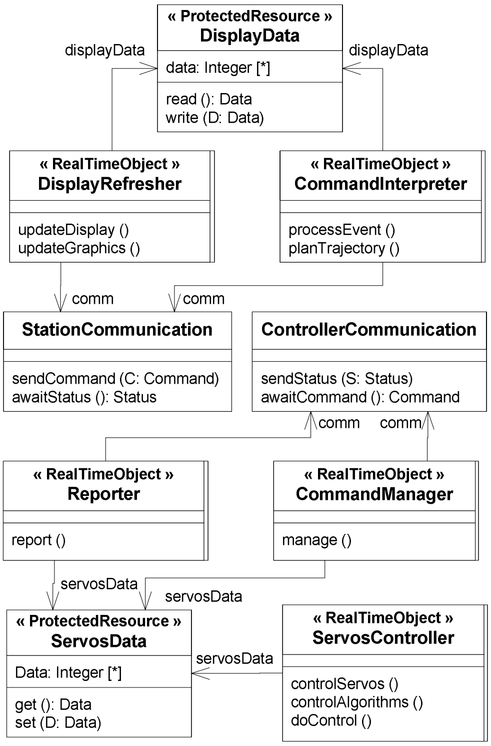


Figure 8. Software architecture of the Teleoperated Robot

To organize the ACCORD|$_{\text{UML}}$ models annotated for schedulability analysis, we adopt the concept of *views*, which represent the concern models composing the SAM's analysis context concept. In this way, we provide separated diagrams for specifying the SAM concepts of *workload situation*, *behavior execution*, and *resources platform*. Next, we show some examples that illustrate this organization.

5.1 Example of Workload Situation Model

The RT_Situation to be analyzed contains three transactions with hard real-time requirements. They all use the processing resources Station, Controller and CAN_Bus and interact by accessing protected objects.

The *Control Servos* transaction executes the Control response with a period and a deadline of 5 ms. The *Report Process* transaction transfers the sensors and servos status data across the CAN bus, to refresh the display with a period and deadline of 100 ms. Finally, the *Execute Command* transaction has a sporadic triggering pattern, but its inter-arrival time between events is bounded to 1 s.

Figure 9 shapes a UML Interaction Overview Diagram (IOD) for the Teleoperated Robot example. This activity diagram represents a *workload situation* model consisting of the three above-mentioned transactions characterized by their *triggers* and *responses*. These three *transactions* explicitly introduce the semantic of concurrency for the *activity partitions*. *Triggers* introduce the semantic of event sequence arrivals for the execution of each *interaction* invocation. We also include some NFP annotations for trigger and responses.

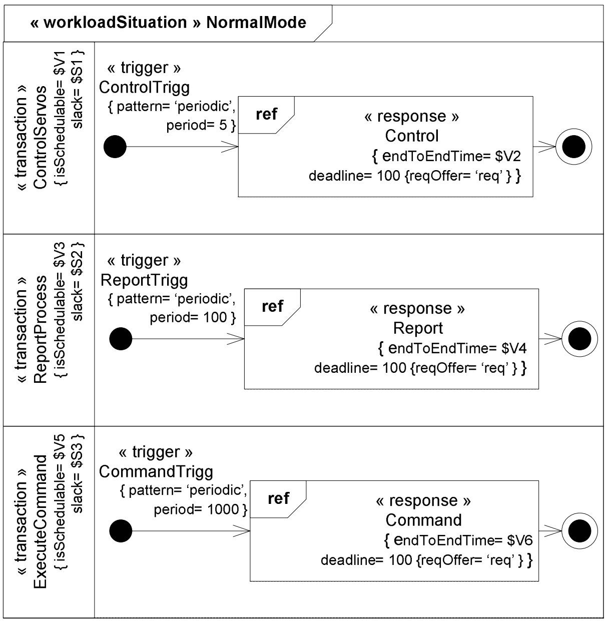


Figure 9. Example of Workload Situation model

5.2 Example of Behavior Execution Model

The *Behavior Execution* concept extends the UML metaclass *Interaction,* which can be modeled by a set of sequence or communication diagrams in UML2. In our example, we applied it to sequence diagrams. Thus, *triggers* extend the metaclass *message*. *SActions* extend the UML2 concept of *execution specification*. Finally, *shared resources* are *lifelines* of the sequence diagrams. The chain of actions (connected by the successor-predecessor patterns) symbolizes the concept of *response scenario*.

In Figure 10, we present one of the three scenarios for the Report transaction. This scenario represents the behavior description of a *response scenario*. Thus, this response scenario is completed with real-time constraints that should be checked in the further schedulability analysis.

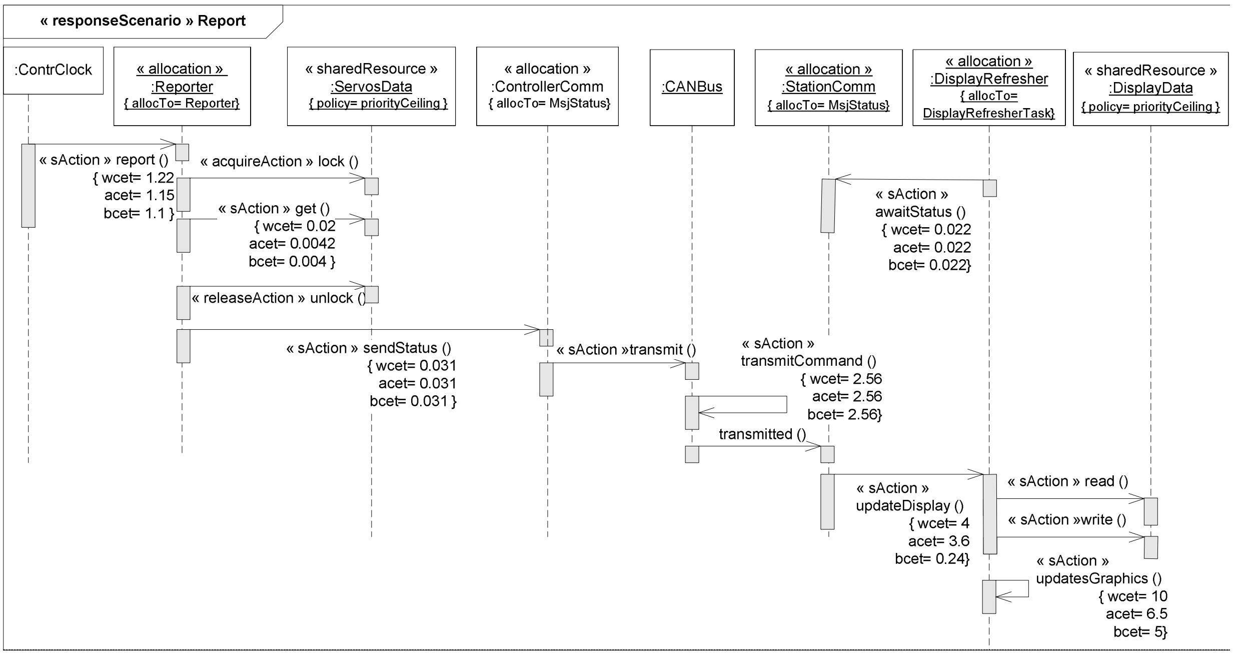


Figure 10. Example of Behavior Execution model

5.3 Example of Resources Platform Model

In Figure 11, the "Execution Engine" domain concept extends the metaclass *nodes*, and the metaclass *Instance Specifications* are used to represent schedulable entities, shared resources, and eventually schedulers.

5.4 Some Schedulability Analysis Results

Table 2 shows the most relevant results obtained from the MAST schedulability analysis tools [10]. In this table, we have compared the end-to-end times of each of the three transactions of the Real Time Situation with their associated responses.

Transaction/Response	Slack	EndToEndTime	Deadline
Control_Servos	101.56%		
Control		3.05 ms	5 ms
Report_Process	189.84%		
Report		39.1 ms	100 ms
Execute_Command	186.72%		
Command		359 ms	1000 ms

Table 2. Results of Schedulability Analysis with the MAST tool

In order to get a better estimation of how close the system is from being schedulable (or not schedulable), the MAST toolset is capable of providing the transaction and system slacks. These are the percentages by which the execution times of the operations in a transaction can be increased yet keeping the system schedulable.

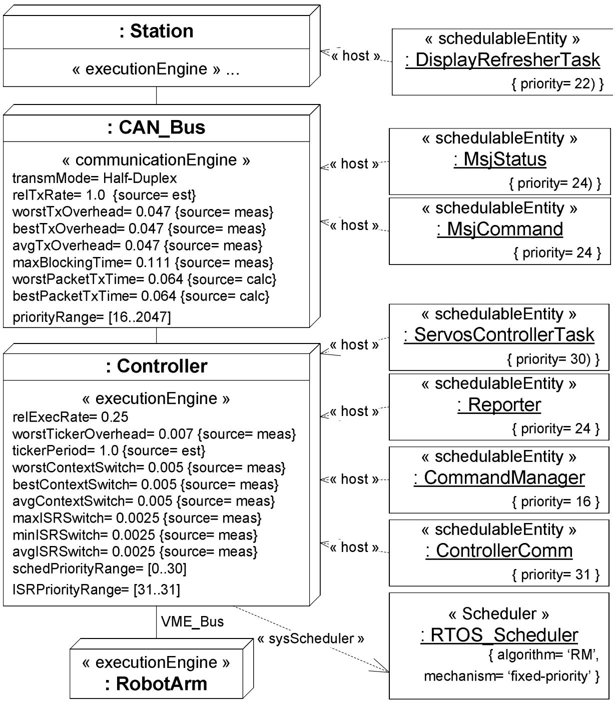


Figure 11. Example of Resources Platform model

Notice that most analysis tools operate on a simplified view of a system, as was illustrated in this example. However, this profile allows annotations and interpretations to be attached at the level of detail desired by the designer. Indeed, even if the specification contains extreme detail, the annotations may optionally be applied to aggregates. This is an overriding reason to find a path to annotations that require a minimum of effort, with a minimum of additions to the design model, and with clear, non-fragmented specifications of NFPs. It is also essential that NFPs can be attached to a real software design, rather than requiring a special version of a design created only for analysis.

6. Conclusions

This paper describes the MARTE schedulability analysis sub-profile for enabling timing predictions. One of the main goals behind this sub-profile is to provide a common framework within UML that fully encompasses the most common schedulability analysis techniques but still leaves enough flexibility for different specializations.

The approach is supported on the NFP modeling framework. It defines a set of mechanisms to declare and specify NFPs that are necessary for different kinds of quantitative analyses. The relationships between NFPs annotations and UML modeling elements is discussed in order to show how to declare domain-specific NFPs, and how to express NFP values attached to model elements.

The schedulability analysis modeling (SAM) framework provides a common modeling basis for different analysis techniques by factorizing concepts that are used by the various schedulability methods. This framework extends the previous SPT's schedulability sub-profile and reorganizes it into generic and consistent modeling concerns (workload, behavior and platform). SAM attempts to enhance the expressive power of UML models and the profile usability by easing the comprehension of the global framework.

The usage of NFP and SAM frameworks are illustrated by the utilization of the proposed annotations on a typical distributed real-time application, formulated in the frame of the ACCORD$|_{\mathrm{UML}}$ methodology. This work can be seen as a first reflection of the UML MARTE profile's schedulability analysis capabilities.

7. References

[1] Object Management Group: UML Profile for Schedulability, Performance, and Time, Version 1.1. 2005. OMG document: formal/05-01-02.

[2] Object Management Group: Pending Issues sent to the OMG Finalization Task Force: UML Schedulability, Performance and Time profile.

[3] S. Gérard (edited by): Report on SIVOES'2004-SPT Workshop on the usage of the UML profile for Scheduling, Performance and Time Mai 25th, 2004, Toronto, Canada.

[4] Object Management Group: UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), RFP. 2005. OMG document: realtime/05-02-06.

[5] Object Management Group: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. 2004. OMG document ptc/04-09-01.

[6] B. Selic, A Generic Framework for Modeling Resources with UML. IEEE Computer, Vol.33, N. 6, pp. 64-69. June, 2000.

[7] S. Gérard: "Modélisation UML exécutable pour les systèmes embarqués de l'automobile", PhD Thesis. 2000, Evry, Paris.

[8] Object Management Group: Systems Modeling Language (SysML) Specification, Version 0.9. Draft. 2005.

[9] T. H. Phan: "Analyse d'ordonnançabilité d'applications temps réel modélisées en UML", PhD Thesis. 2004, Evry, Paris.

[10] J.L. Medina, M. G. Harbour, and J.M. Drake: MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems. Proc. of the 22th IEEE Real-Time Systems Symposium, pp. 245-256. December 2001.

[11] R. Chen, M. Sgroi, G. Martin, L. Lavagno, A. L. Sangiovanni-Vincentelli, J. Rabaey: UML for Real: Design of Embedded Real-Time Systems, Edited by B. Selic, L. Lavagno, G. Martin, pp. 189-270, Kluwer Academic Publishers, May 2003.

[12] Object Management Group. MDA Guide Version 1.0.1. 2003.

[13] Object Management Group. Unified Modelling Language: Superstructure Version 2.0. 2004. OMG document ptc/04-10-02.

[14] S. Graf, I. Ober, I. Ober: A real-time profile for UML. STTT, Int. Journal on Software Tools for Technology Transfer Springer Verlag. 2004.

[15] K. Tindell: Adding Time-Offsets to Schedulability Analysis: Technical Report YCS 221, Department of Computer Science, University of York, January 1994.

[16] Sha, L., Abdelzaher, T., Arzen, K., E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A., K.: Real Time Scheduling Theory: A Historical Perspective: Real-Time Systems Journal, Vol. 28, No, 2-3, pp. 101-155, ISSN:0922-6443, November-December 2004.

[17] H. Espinoza, H. Dubois, S. Gerard, J. Medina, D.C. Petriu, M. Woodside, "Annotating UML Models with Non-Functional Properties for Quantitative Analysis", Proc. of MoDELS'2005 Satellite Events, Lecture Notes in Computer Science, Springer, 2005.

[18] Object Management Group: UML Profile for Modelling and Analysis of Real-Time and Embedded systems (MARTE), Initial Submission: ProMARTE team. 2005. OMG document: realtime/05-11-01.

[19] E. Colbert, "Overview of the UML Profile for the SAE AADL" (presentation): http://la.sei.cmu.edu/aadlinfosite/AADLPublications&Presentations.html, SAE World Aviation Congress Nov 2004.

[20] The ProMARTE home page: http://www.promarte.org

A3S method and tools for analysis of real time embedded systems

S. Rouxel, G. Gogniat, J-P. Diguet,
J-L. Philippe
LESTER. CNRS FRE 2734
University Research Laboratory
France
<rouxel, gogniat, diguet, philippe>@univ-ubs.fr

C. Moy
SCEE Group, SUPELEC
Cesson-Sévigné
France
christophe.moy@rennes.supelec.fr

Abstract

This paper describes a fast prototyping tool targeting software radio applications. It is based on the Unified Modeling Language (UML) and combines a Software Defined Radio UML profile to implement an MDA approach within EDA tools for multi-level verifications from type compatibility to schedulability analysis and memory use rate over an heterogeneous platform. Our approach relies on performance analysis to improve architecture and application matching thanks to non-functional criteria. The main contributions of our work are the improvement of the original meta-model of the Software Radio UML profile and its integration within a unified design framework. From a high abstraction level of a software application we perform extensive verifications and analysis to validate the designer hardware architecture choice and the corresponding implementations.

1. Introduction

Complex System on Chip (SoC) challenge is now achievable since both required hardware resources and integration technologies correspond to reality. The telecom domain is a great example where the SoC paradigm already enables the design of multi-standard chips (e.g. GSM, IEEE 802.11, IS-95). Such an evolution promotes the Software Radio concept for the management of multiple standards [1][2]. However, the design of such systems based on heterogeneous platforms (e.g. DSP, FPGA, GPP, memory) and intensive-computation software applications (e.g. encryption, scrambling algorithm, and service management) cannot anymore be addressed with traditional Electronic Design Automation (EDA) tools. Actually higher levels of abstraction are required to cope with the design complexity and to provide the designers with an early feedback. Such co-design tools partly exist and are based on scalable hardware and software IP reuse. Some of these can already meet the design constraints, like CoWare, that uses SystemC/C++ hardware language specifications, or CoFluent studio, that is based on the MCSE methodology (Co-design Methodology for Electronic Systems) [3][4]. However regarding the current initiatives our approach is original in the way that we combine a Software Defined Radio (SDR) UML profile to implement an MDA approach within EDA tools for multi-level verifications over an heterogeneous platform. Furthermore we have defined very precise models through the A3S profile to perform accurate performance evaluations at the first stages of the design flow. In this paper we present our unified way to fill the gap between the specification and the prototyping phases by using UML. Our work is illustrated through an UMTS transceiver case study.

Major projects related to software radio are described in UML which enables modeling systems through a graphical approach. Furthermore UML continuously evolves to consider new specific characteristics from different activity domains thanks to the development of new profiles. A profile extends the UML language for a work context, which offers scalability. It specifies all characteristics (e.g. elements for real-time application) and relations between the UML elements. It allows model-based a priori verifications. A designer relies on the profile to analyze, generate code and specify various application and architecture constraints. Moreover, dependencies, inheritance, or groupings between profiles can be performed to promote the reuse of domain specific needs. Regarding the software radio application, three profiles are of interest: UML profile for Software Radio [5], UML profile for Schedulability Performance and Time [6] and UML profile QoS and Fault tolerance [7]. Each profile brings out some specific characteristics that are useful to perform the evaluation of the system performances. Dealing with these profiles, a system can theoretically be accurately specified by integrating various constraint types (e.g. power consumption, bounded execution time).

However these profiles partially address the parameters required for SDR prototyping. Our work proposes to extend their coverage through the development of a new and specific one. Its purpose is to highlight standard concepts required for system prototyping and to add hardware attributes that are not currently taken into account for Software Defined Radio applications. Furthermore the goal of our A3S project (System Application Architecture Adequacy) is not limited to the definition of the A3S profile but also targets its implementation within a rapid-prototyping tool to evaluate the feasibility of complex applications over heterogeneous platforms (with DSP, FPGA components). Specification of dynamic reconfiguration is also investigated since this feature will be mandatory especially for Software Radio applications.

The remainder of this paper is the following. Section 2 presents various high level system specifications and most relevant tools relying on the MDA approach. Section 3 provides a global approach of system modeling as promoted within our project. Section 4 details the A3S profile and the UML modeling by giving the set of parameters required to compute verifications and performance evaluation. Section 5 details the scheduling analysis technique and the approach for design space exploration. Section 6 gives an example of an UMTS application modeling. Section 7 concludes the paper and gives an overview of future work.

2. Related Work

Many tools aim at modeling systems, performing verifications, simulations, validations, and synthesis. Different modeling styles with different granularities are considered, different input specification languages as C, SystemC, VHDL, are also used to validate, verify, simulate or emulate a system [8][9]. First co-design tools, like VULCAN are using simple and limited hardware architecture models, others like COSYMA are based on dedicated hardware co-processors to speed up software execution [10][11]. COWARE and PTOLEMY consider heterogeneous specifications to respectively design specific applications (embedded telecom) and co-simulate heterogeneous HW/SW systems [12]. However these approaches are limited as they require the use of different tools that must be kept updated. Actually the goal is to perform both modeling and design specification of hardware platform and software application within a single tool and through a common language to be less dependent of multiple software update [13]. The SoC Environment (SCE) developed within the University of California, Irvine provides such an approach as the design specification within each stage of the design flow is defined through a SpecC code [14]. However, the use of a generic language, common to different domains, that is enough flexible to model all co-design aspects (architectural and application specifications, component properties, constraints specification) will be mandatory to accelerate the design cycle and to promote the design reuse. To target such a philosophy, the most recent rapid prototyping tools integrate methodology of hardware-software co-design into the concept of MDA (Model Driven Architecture) through UML.

ZeligSoft proposes a code generator that produces Software Communications Architecture (SCA) artifacts for Corba compliant targets. This approach is sizable regarding different aspects such as the SCA core framework [15] but no SoC meta-model is provided. In [16] the authors focus on the deployment design step but the analysis method is limited to Interface Definition Language (IDL), type compatibility and pure software concerns. There is no analysis addressing embedded systems issues such as memory, bus, real-time, power.

The Prompt2Implementation targets an MDA for SoC design. It is based on the ISP UML profile [17] for parallelism expression at task and data levels and on model to model engines. The main objective is to produce a simulation code (e.g. SystemC TLM) based on mapping rules. This is a very ambitious project restricted to very intensive signal processing, but the tools seem to be under development. Moreover, this approach does not address the SDR concept. The association between UML and SystemC is a promising approach, which is also explored in [18]. In this work, a UML SystemC profile is proposed and used to generate SystemC code. An object-oriented HW/SW synthesis flow based on an UML initial specification is described in [19]. The MOCCA compiler implements an MDA approach based on system, platform and deployment models. The current implementation is based on a processor/FPGA platform where SW and HW components have been implemented. This work is interesting but does not rely on SDR UML profile. In [20] the authors present a framework for software design space exploration based on performance and power estimation issued from an UML specification. The method is based on a pre-characterized platform and enables the evaluation of software implementation solutions specified by the designer. FZI is developing a framework for the communication conflict analysis in a SoC context. In this approach [21] UML and SysML are combined to specify architectures when on the other side a sequence diagram is used to specify the application. After refinement and component mappings a conflict graph is built to analyze communication scheduling. Finally, the UML2.0 profile for SoC is another initiative that reached the approval step in September 2005. No tool is currently proving the concepts that are located at a software level.

Compare to previous efforts our approach relies on our UML A3S profile that inherits from other standardized profiles and extends them. This profile improves and offers more hardware specification possibilities that are essential for software radio or other electronic systems in order to specify hardware and software architecture systems. In addition our high abstraction level specification alleviates the modeling and the validation of applications that belong to other specific application domains. Moreover, as we consider applications as a set of IPs, components are only characterized by non-functional parameters instead of source codes (which depend on their implementations and need different tools).

3. A3S Design Approach

A3S approach proposes a UML software framework where the designer can rapidly and easily prototype his system and check if constraints are met in terms of timing, memory, area, and power consumption [22]. The main steps of our design flow for virtual prototyping are depicted in Figure 1.

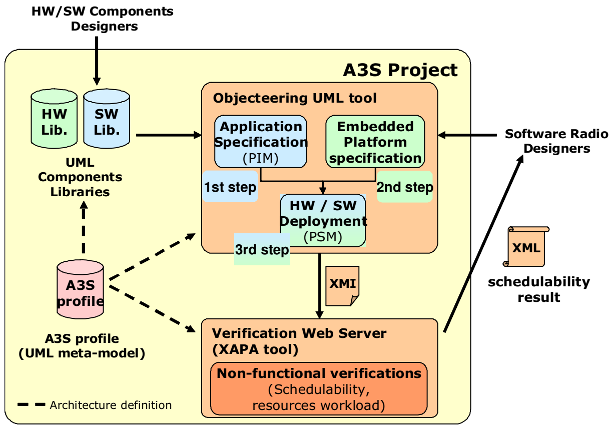

HW/SW Components Designers
A3S Project
HW Lib.
SW Lib.
UML Components Libraries
Objecteering UML tool
Application Specification (PIM)
Embedded Platform specification
1st step
2nd step
HW / SW Deployment (PSM)
3rd step
Software Radio Designers
XML
schedulability result
XMI
A3S profile
A3S profile (UML meta-model)
Verification Web Server (XAPA tool)
Non-functional verifications (Schedulability, resources workload)
Architecture definition

Figure 1. A3S design flow

One important question is to know who the final user of the tool will be. Two kinds of actors can take benefit of the A3S framework:

- The component designers. They are concerned by the components definition (HW and SW) which takes place within the modelling and specification tool. They will create the software and hardware components libraries (IPs). For this kind of actors, some ergonomic wizards included in the design tool will help them to provide the correct values when creating new hardware and software components.
- The software radio designers. Their goal is to design and tune the software radio platform and waveform. For this kind of actors, other ergonomic wizards will help them to instantiate and place the A3S software radio components with conformity to the A3S profile. They will be able to manually perform several HW/SW deployments in order to reach an optimized solution.

The verifications performed by the tools are related to the A3S profile (see Section 4). They allow the designer to see in a simple glance the errors within his design during each step of the A3S design flow. It is always possible, in spite of the existence of the GUI, that the designer gives values that are not coherent. Thus, extensive verifications enable a faster and safer design flow. Some errors can be related to the architecture of a platform, or the connection between the software application and the embedded platform. The designer can perform the verifications for the main points of a design (libraries of hardware components and software components, hardware platform and software application) or for a whole project.

Each step of the design flow is now detailed in the next sections.

3.1. Application specification (1st step)

With the MDA approach, software application and hardware architecture can be specified independently, so 1st step and 2nd step (see Figure 1) can be exchanged. To manage complexity, an application is split into several functions that are represented by independent generic software (SW) components. This view corresponds to PIM (Platform Independent Model) since each function can be potentially mapped onto any hardware component. SW components have specific non-functional parameters that correspond to specification constraints coming from the application or from some designer requests. An example of these parameters is the periodicity of the SW component which is independent from any implementation. More information about these parameters is detailed in Section 4. At this stage of the design flow SW components can represent any function.

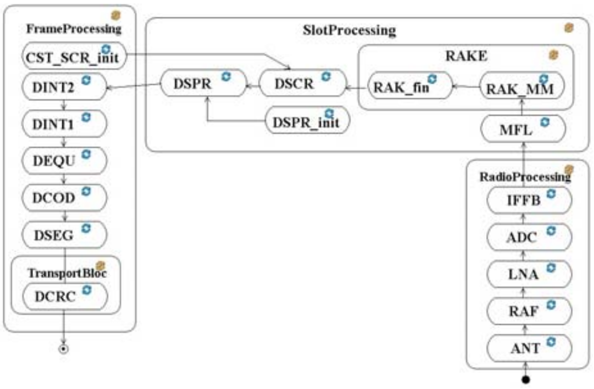


Figure 2. UMTS-FDD Receiver Activity Diagram

The application is modeled through a functional scheme based on the UML Activity Diagram which is composed of a set of ActionStates (SW components) and transitions. Transitions correspond to dependency relations between functions and have specific parameters related to the exchanged data (e.g. number, size). For each component, the designer specifies the corresponding parameters value. An Activity Diagram has been considered since it enables the description of the dynamicity of a system. Activity Diagrams allow the modeling of the process described by activity chains with information related to transmission, connection management, and activity responsibility description.

An activity diagram example for an UMTS-FDD receiver is given Figure 2. This diagram also addresses the links between the different SW components to specify the system radio functionality. The black dot represents the input of the application which takes place at the propagation channel side. Each arrow corresponds to an edge (transition) and represents a data-flow dependency. The UMTS-FDD receiver is mainly a data-flow application with periodic and iterative functions (FrameProcessing, SlotProcessing, RadioProcessing, TransportBloc). The black dot in the circle is the output of the application; it corresponds to the exchanged data between the physical layer and the higher layers of the OSI model.

Through this model the designer can easily replace, add, move/remove a SW component, or modify some parameters to enhance the algorithm and thus test various configurations. By this way, he can analyze the impact of different reconfigurations, which is of major importance in a software radio context. Once the application model is completed, some coherency constraints verifications are performed. Among them, the tool verifies that all connections between SW components have been correctly done, through compatible data format and that all required parameters have been settled. These verifications have been implemented within the Objecteering case tool [23].

3.2. Embedded platform specification (2nd step)

This step deals with the platform specification. Each hardware component is described in a hardware library (DSP, FPGA, GPP, memory, interconnect and ASIC) corresponding to an UML package. Each component has specific attributes defined through its stereotypes (this point is developed in Section 4). The designer builds his platform by assembling hardware component instantiation (in UML sense) through a UML deployment diagram. Many hardware platforms can be realized, especially heterogeneous platforms. This kind of architecture is essential for telecom applications like software radio that need flexibility (offered by FPGA and DSP components for hardware and software reconfiguration) and important computation resources (multi-processor). A Deployment Diagram has been considered since it enables the description of the physical connections that exist between the hardware devices located on the platform.

3.3. Hardware/Software deployment (3rd step)

After the software application and hardware platform modeling steps, the designer chooses which dedicated SW component is implemented onto which hardware component. For each SW component, the designer selects the corresponding function in the software component library as a SW component corresponds to a processing element that is not dedicated to a specific target (PIM). Thus, the function represents an implementation of the SW component onto a processor (e.g. DSP, GPP, µC), a FPGA or an ASIC. The target hardware component selected to implement the SW component is obtained by defining an instance of a hardware component within the hardware platform in the UML deployment diagram.

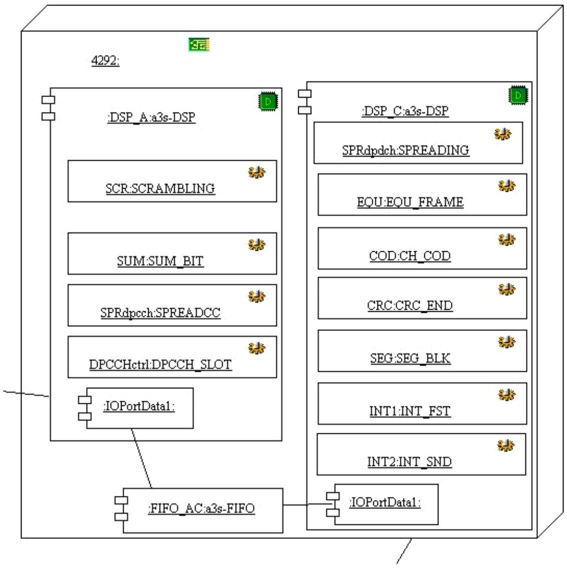
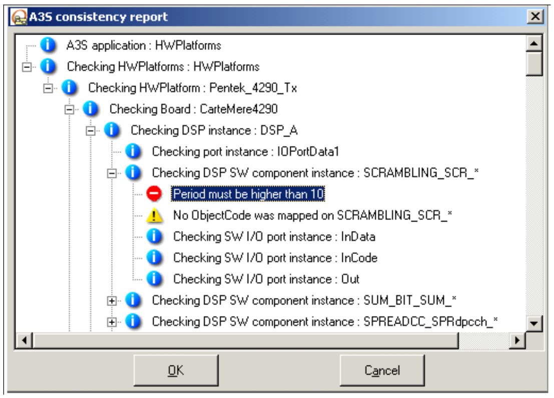


Figure 3. Deployment diagram after mapping

When a SW component has been deployed onto an hardware resource new attributes are highlighted which represent the implementation details. This refinement corresponds to the transition from a PIM to PSM (Platform Specific Model) model where specific parameters are requested to do a performance analysis. A broad range of implementation solutions can be tested for a specific platform due to all possible combinations. The example in Figure 3 depicts an hardware platform composed of two DSPs (DSP_A, DSP_C) on which different software components are implemented (e.g. scrambling function is implemented on DSP_A). Thus the deployment diagram is refined by a software component instantiation implemented into a hardware component instantiation. This partitioning is performed through links between the software components from the UML activity diagram and the hardware components from the UML deployment diagram. For example, the DSP_A that is connected to DSP_C via FIFO_AC handles four functions (SCR, SUM, SPRdpcch, DPCCHctrl).

3.4. Non-functional verifications

During the specification steps, non-functional verifications are automatically performed thanks to the use of the A3S meta-model. Verifications within the Objecteering case tool are stored in a tree which allows the definition of priority levels for all non-functional verifications. Methods related to the verification rules are connected to the branches and the leaves of the tree. Verifications can be simple numeric value checking (attribute value different from zero) or more complex techniques to verify specific properties (data production and consumption coherency). All the methods are coded using the J language which is dynamic and interpreted. J language exploits the meta-class, meta-associations and meta-attributes to get access to the attributes which are stored as taggedValues. The verification report is generated while traveling through the tree and displayed to the designer (Figure 4).



```
A3S application : HWPlatforms
Checking HWPlatforms : HWPlatforms
  Checking HWPlatform : Pentek_4290_Tx
    Checking Board : CarteMere4290
      Checking DSP instance : DSP_A
        Checking port instance : IOPortData1
        Checking DSP SW component instance : SCRAMBLING_SCR_*
          Period must be higher than 10
          No ObjectCode was mapped on SCRAMBLING_SCR_*
          Checking SW I/O port instance : InData
          Checking SW I/O port instance : InCode
          Checking SW I/O port instance : Out
        Checking DSP SW component instance : SUM_BIT_SUM_*
        Checking DSP SW component instance : SPREADCC_SPRdpcch_*
```

Figure 4. Checking procedure of a SW application after deployment – first part (PIM) – second part (PSM) for each HW platform

3.5. Schedulability analysis results (4th step)

Results are provided through a schematic view defined in a UML sequence diagram which is close to a Gantt diagram. The results emphasize the performances achieved for a heterogeneous platform with multi-processor resources to perform the application. For example, execution time, resources use rate, system evolution (scheduling), allocated memory resources are exhibited. Scheduling information is very important as if the system cannot be scheduled or if it does not reach the required timing constraints, the solution is not relevant.

If the solution does not satisfy the constraints, it is easy to modify the implementation choices just by modifying the links between software and hardware components in the UML activity diagram without modifying the diagram. As several applications and platforms can be specified it enables testing an application onto different platforms and with different implementations for a same platform. It also promotes testing different configurations and re-configurations of the system. The design space exploration is performed manually and iteratively in the current methodology. It is also possible to modify some hardware characteristics by changing hardware component parameters value. Moreover, this tool returns results that help designer to perform modifications according to identified critical functions.

3.6. Ergonomics – wizard – GUI

To provide an intuitive verification tool, the checking preserves the hierarchy of the elements within a project (components, application/platform, deployment) and indicates through a message the possible errors or warnings. Thus, it is easy for the designer to analyze where the problem comes from and to further help him the tool points out the element affected by the error when clicking on an error message.

Figure 4 shows the consistency report as it is provided to the designer. As we can see an error is highlighted which enables the designer to change his specification before going through the non-functional verifications tool that analyzes the schedulability of the system.

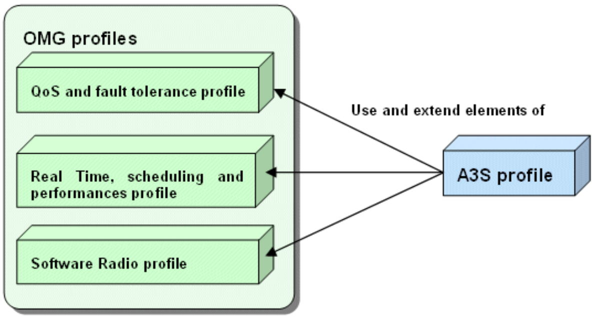

OMG profiles
QoS and fault tolerance profile
Real Time, scheduling and performances profile
Software Radio profile
Use and extend elements of
A3S profile

Figure 5. Relation between A3S profile and the OMG standard profiles

4. A3S profile and UML Modeling

4.1. A3S Profile

One of the goals of A3S is to emphasize non-functional characteristics on PIM and PSM and to analyze them during the verification phases. Currently, major software radio projects are described by UML class diagrams for architectures and sequence diagrams for applications. The UML profile for software radio proposes a set of PIM and PSM stereotypes to describe platform independent or dependent architectures of radio systems from a functional point of view. To allow the precise definition of signal processing applications, it can be extended through the use of stereotypes coming from the QoS profile and the Real Time Scheduling and Performances profile on each of the components addressed by the Software Radio profile.

In order to address the DSP/FPGA specific domain, it is possible to extend the software radio profile by introducing specific DSP and FPGA stereotypes representing DSP and FPGA components derived from the processor stereotype of the software radio profile. These new stereotypes will be tagged with stereotypes extracted from the QoS profile to describe the quality of service metric of the DSP and the FPGA.

Using standardized profiles and the components they introduce, will allow the designer to reuse some legacy components by wrapping them into a standard component exhibiting the compliant interfaces. It will enable the designer focusing on the architecture or the system composition, instead of being compelled to discover and/or create new components from scratch. This method is already used for a long time by software developers to de-couple from third-party provided components. Such an approach is the only way to enable a smooth transition from existing methods to new ones. It also allows the integration of non-compliant external component.

The A3S profile formalizes through a rigorous semantic the elements that will be used to build the software radio architecture models. Our formalization enables the definition of the verification rules. These elements extend or use some elements extracted from the previously explained OMG standard profiles, as illustrated in Figure 5.

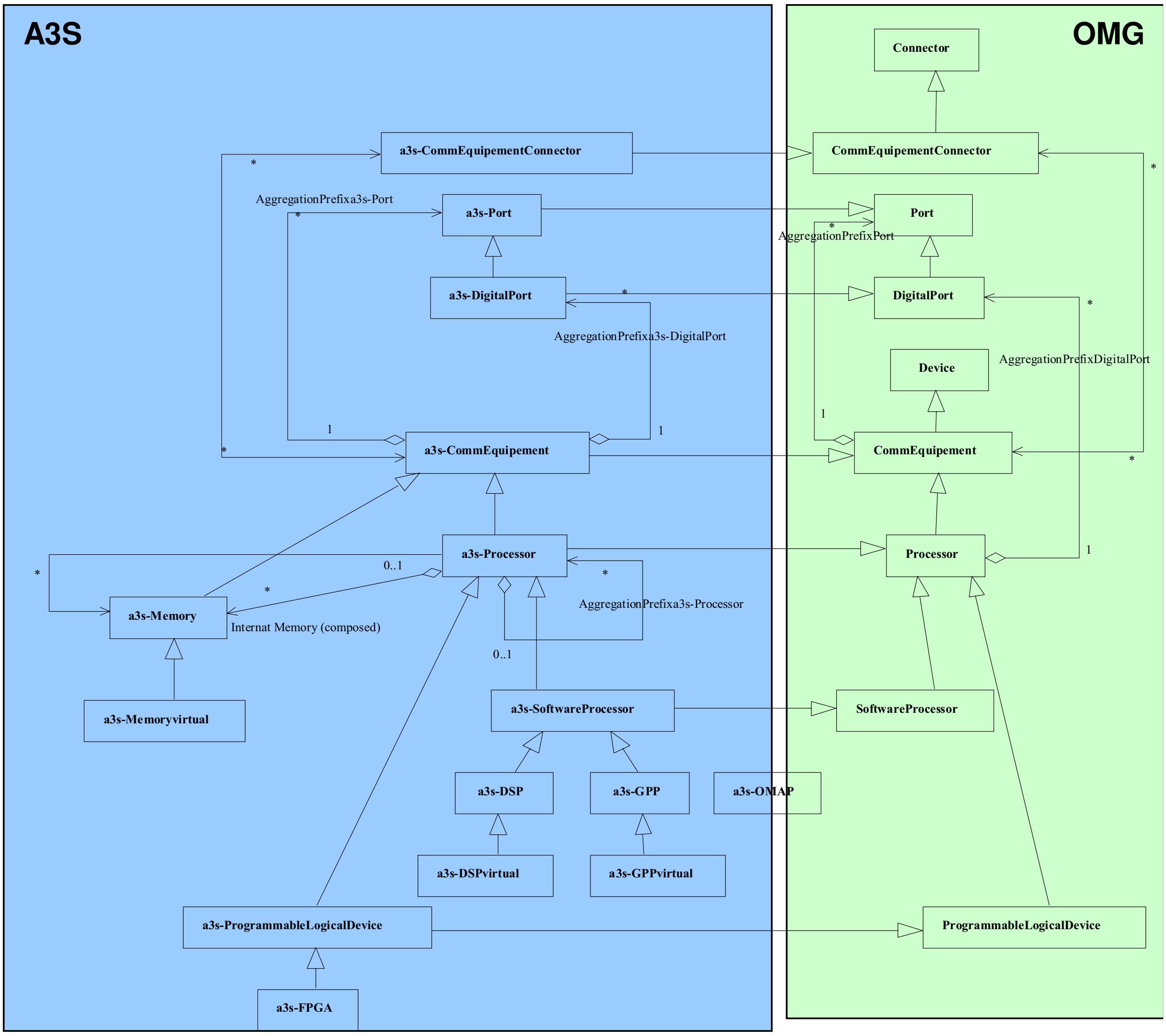


Figure 6. A3S profile's hardware meta-model

This warranties the timelessness, the interchange and the reusability of the A3S models. Since the interfaces can be standardized by this way, it is then possible to work and verify any A3S model assuming that the tools have the A3S profile. This A3S profile main interest resides in the fact that all the interfaces may be standardized, and that all the elements are redefined from the basic types, warranting an automatic generation of interface specification through the IDL syntax language.

Figure 6 illustrates the hardware meta-model of the profile that defines the stereotypes that will be used to design Software Radio platforms. It extends the OMG software radio model (on the right part of the figure), by defining new stereotypes prefixed by the "a3s-" keyword. They inherit from each of the main components of the OMG software radio profile and provide some non-functional information (on the left part of the figure). For instance, according to the OMG software radio profile, each hardware component of a software radio can be stereotyped by the CommEquipment element and that the CommEquipment are connected to each other through some CommEquipmentConnectors linked to their DigitalPort. A3S provides the same elements extended with QoS characteristics, that may range from data size and processing frequency to power consumption. Such an inheritance is generic enough to envision the future addition of new QoS characteristics to an element of the A3S profile, without disturbing all the models of the software radio platform.

The first step for QoS definition of software radio elements is to specify the QoS language that will be used during the modeling phase. For our purpose, it will allow the specification of a particular kind of software radio component, the fields that are relevant to quality of service and that must be filled with accurate values in the PSM model. The definition of such a QoS language specific to the A3S issues is performed using the QoSCharacteristic elements of the QoS and Fault Tolerance profile. QoSCharacteritics can be extracted directly from the catalog of well known QoSCharacteristics of the QoSProfile, but can also be defined from scratch, inherited from other QoSCharacteristics or aggregated by others. The set of QoSCharacteristics obtained by this way, is then stored in a QoSCatalog dedicated to the A3S needs. At the design time, these QoSCharacteristics will be implemented into QoSValues which will be applied to PSM software radio components. Figure 7 illustrates the definition of the QoS characteristic of an FPGA, and Figure 8 illustrates how it is possible to describe the QoS offered by the platform specific FPGA-pentek-3292.

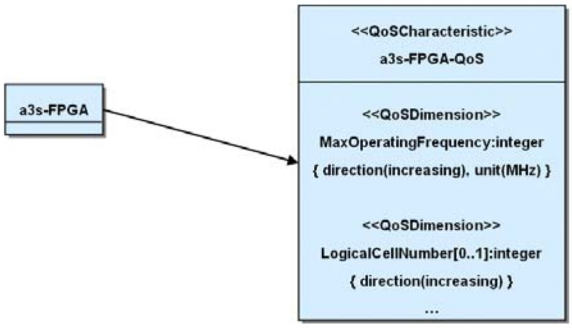

a3s-FPGA
<<QoSCharacteristic>>
a3s-FPGA-QoS
<<QoSDimension>>
MaxOperatingFrequency:integer
{ direction(increasing), unit(MHz) }
<<QoSDimension>>
LogicalCellNumber[0..1]:integer
{ direction(increasing) }
...

Figure 7. Definition of the QoS characteristic of a FPGA

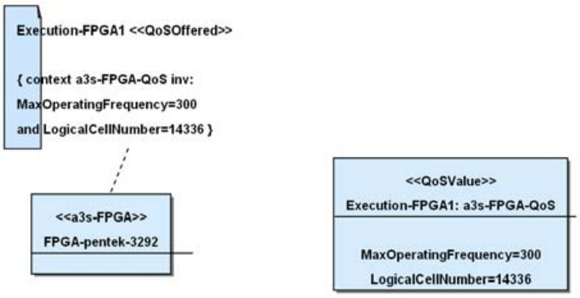

Execution-FPGA1 <<QoSOffered>>
{ context a3s-FPGA-QoS inv:
MaxOperatingFrequency=300
and LogicalCellNumber=14336 }
<<a3s-FPGA>>
FPGA-pentek-3292
<<QoSValue>>
Execution-FPGA1: a3s-FPGA-QoS
MaxOperatingFrequency=300
LogicalCellNumber=14336

Figure 8. Definition of the QoS value that a specific FPGA may have

4.2. UML Modeling

During the application and platform specification steps, the designer provides the values of the software and the hardware component attributes to perform the coherency verification and schedulability analysis of the system. Each component (software and hardware) can be characterized in three parts as described in Figure 9. First part corresponds to the non-functional characteristics (attributes) of the component.

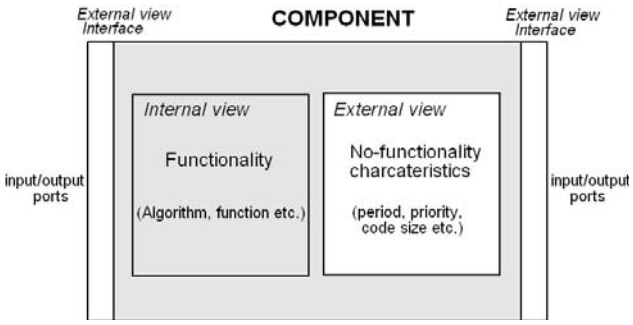


Figure 9. Component Views

For software components it represents the temporal aspects of the function (e.g. period) and the data characteristics. The second part describes its interface (its I/O port) with significant attributes relative to exchanged signals. The third part is relative to the functionality of the component. For hardware components it corresponds to the clock frequency, the type and quantity of internal/external memory. This view mainly corresponds to specification constraints. As our approach relies on IP cores, the internal view of the component is not explicitly represented since we assume that IP cores functional behavior (C, C++, SystemC, VHDL) is validated through other means that are not in the scope of this paper. In our case attributes can be provided using the IP characteristics.

UML stereotypes permit to identify and characterize any element by assigning different parameters called "attributes". So each element of UML can be specialized by using different stereotypes that are used to define the component parameters.

Generic SW components which are not yet implemented have different attributes (e.g. a function is periodic or not, it has an initialization part or not) compared to a dedicated SW component which represents one implementation choice of a generic SW component. Each implementation choice adds some specific constraints that are highlighted through the non-functional attributes. They deal with function periodicity, execution time, code size, priority level if a RTOS is used, and other attributes like data/code localization, and access memory types. HW components have different stereotypes, which lead to the difference between HW processing components (DSP, ASIC, processor), memory components (FIFO, RAM, ROM), reconfigurable components (FPGA) and communication components (Bus, wire). Specific performance parameters are considered according to the hardware component (frequency, data/program memory size, port type, data width, throughput).

All the parameters are required to perform the performance analysis. They are used during the scheduling analysis step (see Figure 1), to compute resources use rates, to perform constraints verification and to check the coherency of the system.

4.3. XAPA tool for schedulability analysis

Once specification and mapping have been completed and coherency verifications have been performed (i.e. no error about HW/SW connection, all attribute settled), the A3S tool generates a XML file gathering the information about the system. The file contains the diagrams (activity, deployment), the hardware/software component allocated, and the attributes value. More precisely the UML activity diagram that represents the functional application scheme of the system is encompassed in the XML file. Thanks to an XML parser this diagram is converted into a task graph. Each ActionState becomes a task and each Transition becomes an edge between tasks. The underlying execution model of both the activity diagram and the tasks graph is data flow. The parsing of this file enables building a General Task Graph (GTG) based on the Radha Ratan model since we consider the corresponding method to perform period derivation [24]. This method computes the period of each task within the GTG even if some are previously unknown. The GTG nodes represent tasks (functions), and the GTG oriented edges are channels from producers (tasks) to consumers (tasks). Each task can be triggered by a data. Each edge contains producer and consumer information corresponding to data to be exchanged between functions. For applications implemented onto multi-processor, functions implementation can lead to additional communication tasks (in case of two tasks connected to each other and implemented on different hardware devices). The period derivation step is performed to compute the timing constraints (periods) that have not been settled by the designer during the specification steps. This point is important, since this kind of computation is very error prone and can be efficiently done with our tool.

The GTG obtained from the XML is then used with the HW architecture characteristics within our real time analysis tool RTDT [25]. This tool performs automatically complex scheduling verification and provides performance analysis results which help designer to drive his choices. Such automatic bridges and tools are essential to improve time to market and quality designs.

5. Real time analysis tool (RTDT)

5.1. Real time scheduling strategy

5.1.1. Task classification

Usually, real-time embedded systems require a simple and safe scheduler which can guarantee that critical aperiodic or periodic tasks meet their deadlines. For these reasons, a static HPF (High Priority First) scheduling policy has been adopted, where the fixed priorities are computed as the inverse of the task period. The worst case response time is computed with an exact analysis [26] or is provided from the library of IPs.

In a first approach we consider two kinds of tasks. The first category is composed of the periodic tasks that are scheduled by means of hard real time constraints (RTC), and sporadic tasks with hard RTC. Like in [27], we consider the sporadic tasks as periodic tasks with a period equals to the minimum delay between two subsequent executions; this value can be provided by the Radha Ratan tool [24]. Actually we have implemented the techniques of this tool in our UML framework in order to derive unspecified periods from global I/O constraints. These period are specified within the output XML file. The second category includes the non critical sporadic tasks which are handled by a task server with the lowest priority that can be fixed by the designer. The task priority is computed as the inverse of the task period.

The question of multi-rate dependencies is solved by shifting the release time computation as detailed in [28].

5.1.2. Response time computation

The exact response time is computed iteratively with the following equation:

$$\forall T_j \in HP(i), \exists R_i \leq D_i / R_i = (C_i + R_i) + \sum_{j \in HP(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \times (C_j + C_{sw})$$

Where:

- *HP(i)*: is the set of tasks with higher priority compared to task *i* (T_i);
- R_i: is the worst case response time of task *i*;
- D_i: is the execution deadline for task *i*,
- C_i: is the execution time of task *i*;
- B_i: is the longest time that task *i* can be delayed by lower priority tasks,
- P_j: is the period of task *j*,
- C_{sw}: is the context switching;

$$C_{sw} = \delta_0 + \sum_k \delta(k)$$

With:

- δ_0: is the context switching overhead without any coprocessor,
- $\delta(k)$: is the overhead due to the coprocessor *k*.

The context switching overhead is the delay between the preemption of a given task and the activation of another task.

5.2. RTOS overhead

The difficulty is that C_{sw} depends not only on the target processor and on the RTOS and its configuration but also on the number of tasks in the system and on the number of coprocessors. Without coprocessor, the available overhead metric is usually an average value estimated with different task sets. We have defined an accurate model for the RTOS overhead which takes into account the following parameters: initialization of the RTOS, context switching, scheduler, task, semaphore, mutex, mailbox, message queue and flag creation, and post/pend of the previous mechanisms. Many parameters influence the model and they must be clearly defined by the designer to compute an accurate value. As an example the overhead on a task execution due to the RTOS compared to the execution of the task without RTOS is defined by the equation below:

With

$$T_{ps_{exewithOS}} = \kappa + \left\lfloor \frac{\kappa}{P_{tickscheduler}} \right\rfloor \times (\gamma + 4)$$

$$\kappa = \gamma \times \left\lfloor \frac{T_{ps_{exewithouOS}}}{P_{tickscheduler}} \right\rfloor + 4 \times \left(\left\lfloor \frac{T_{ps_{exewithouOS}}}{P_{tickscheduler}} \right\rfloor - 1 \right)$$

γ corresponds to the delay due to a scheduler interrupt. The execution time of a task with an OS is thus increased based on the period of the scheduler tick and depends on the number of time a task has been preempted by the scheduler. The influence of the coprocessor is also related to the number of data and status registers.

5.3. Design space exploration for HW/SW partitioning

5.3.1. Cost function

The cost function takes into account the global area of the SoC and its energy consumption. At a high level of abstraction, only relative estimations can be used for SW and HW IPs and the cost function is used to guide the selection of a reduced set of solutions. In order to eliminate solutions, relative costs are used to evaluate the cost value for a given schedulable solution *S*:

$$Cost(S) = \alpha \frac{Area(S) - MinArea}{MinArea} + \beta \frac{Pw(S) - MinPw}{MinPw}$$

with α + β = 1 and where *MinArea* is the schedulable solution with the minimal area without any power consideration and *MinPw* is the schedulable solution with the minimal power without any area consideration. Note that the area cost influences the power consumption through the static power evaluation. So, the α parameter also acts on the power optimization.

5.3.2. Area Cost

The area cost includes the data and code memory size for software implementations, the area of coprocessors that can be shared by various tasks, the area of hardware accelerators and finally the area of memories added for communications.

5.3.3. Power Cost

The model for power evaluation is much more complex. Firstly, the dynamic power consumption depends on the SoC activity, which is strongly related to the task scheduling and switching. Secondly, the evolution of VLSI technology shows that static power consumption [29], especially in FPGAs, can no more be neglected. Finally, in mobile embedded systems the important metric is the system life span. It means that the energy used must be optimized. However, in our context of periodic tasks the energy optimization is equivalent to the average power minimization over the hyper period. Our power model for an implementation *S* is given by:

$$Pw(S) = Pw_d + Pw_s$$

where: Pw_d is the average dynamic power dissipated during a hyper period T_G and Pw_s is the average static power. The details of the power model are out of the scope of this paper and can be found in [28].

5.3.4. Scheduling analysis

The main difficulty during RT scheduling algorithm is related to the iterative scheduling of tasks worst case response time.

A solution is valid if all tasks meet their deadlines. Contrary to the response time computation, the cost is not iterative and must be evaluated first. Thus the schedulability is computed in a three-step approach to limit the computations of iterative response time. The algorithm first tests if the processor rate is lower than 1. As a second test, the fast rate monotonic analysis (RMA) is performed; it gives a sufficient but not necessary condition for schedulability. Finally if the first tests are valid an exact analysis is performed. Note that the designer can specify the CPU ratio *rs* to be guarantied for the server task.

```
Boolean Schedulable (S)
{
    U = ProcUseRate(S) // Processor use rate
    If (U+rs > 1) // rs: task server CPU ratio
            Return false;
    Else if $U + rs \leq n \times (2^{\frac{1}{n}} - 1)$
            Return true;
    Else {
            For all $T_i$ by Decreasing Priority Order
            $R_i$ = ExactResponseTimeAnalysis($Ti$);
            If $R_i > P_i$ Return false;
            Else Return true;
            }
}
```

Figure. 10. Schedulability test

6. UMTS FDD Case Study

The A3S profile has been created to specify the software defined radio physical layer. An UMTS FDD channel in uplink mode has been chosen as a first reference to determine which software and hardware components should be included in the software and hardware components library. This application has also been tested to validate the A3S tool. The UMTS transmitter and receiver applications have been modeled through two different activity diagrams (UMTS receiver modeled in Figure 2). The chosen hardware platform corresponds to a standard board composed of multiple DSPs connected to FPGAs via FIFO and SDRAM memories. The deployment diagram represents the Pentek board (4290) described thanks to the hardware components from the A3S hardware components library (an overview is given in Figure 3). The partitioning has been determined manually by the designer. After specifying the hardware components attributes (using the Pentek board components characteristics), and the software components attributes (using software IP core characteristics), the validation and the schedulability performance analysis steps are performed.

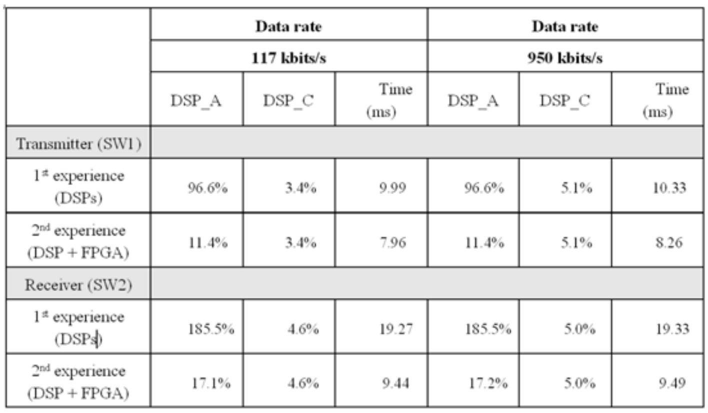
Non-functional attributes are first checked to verify the system coherency. Then the A3S tool generates the GTG file (.gtg) and provides the HW architecture characteristics to perform the schedulability and the power consumption analysis. Our real time analysis tool has been first designed for mono-processor architectures with hardware accelerators; it has been modified to support schedulability analysis in the context of multi-DSP/Processor architectures.

To prototype the UMTS FDD transmitter and receiver we have considered a hardware platform composed of four TMS320C6X DSP running at 300Mhz. Each DSP is connected to a XILINX Virtex XC2V3000 FPGA running at 100Mhz. Each DSP is also connected to an external shared SDRAM memory. The two UMTS FDD software applications, transmitter (SW 1) and receiver (SW 2) are implemented into the hardware platform described above. SW 1 is composed of 11 functions (pulse shaping, scrambling, coding, spreading, integrating …) and SW 2 is composed of 14 functions (matched_filter, rake, descrambling, despreading, decoding …).

Different implementations are considered to verify and validate the efficiency of the A3S tool. These experiments have been iteratively performed in order to meet both architectural and application constraints. The first experience consists in implementing all the functions for SW 1 and SW 2 into the DSPs (software solution), and then in modifying the data rate frequency to see the limits of such a solution. The second experience consists in partitioning the functions implementation between DSPs (DSP_A, DSP_C) and their respective associated FPGAs (FPGA_A, FPGA_C). The critical functions within each application are implemented into the FPGAs and the remainder into the DSPs. For both experiences, the two different data rates (117 kbits/s and 950 kbits/s) are tested. The UMTS design under consideration is not a complete fully realistic UMTS system but contains enough processing element to evaluate the tool.

For each experience, a possible scheduling was determined and the corresponding hardware components utilization rates were computed. The results for one radio frame are given in Table 1. An overall 100% means that all the processing power of all HW devices is necessary to run the application in real time. Less than 100% means that real-time is also reached. When the workload is more than 100% it means that a single HW device (DSP or FPGA) is not enough to run the application. In that case it is necessary to provide a new partitioning to be able to reach the constraints.

In the UMTS standard, a radio frame must be computed every 10 ms. In this first experience, we only consider the execution time issue. It shows that software-only solution is adequate for SW 1 as the rate is correct (<100%) for the two configurations (117kbits/950kbits). Actually this solution is not correct for the 950kbits configuration since the timing constraint is not respected as the execution time exceeds 10 ms (10.33>10). In the case of SW 2, the software-only solution cannot be realized because of the DSP overload (185%). Thus to respect both the DSP workload and the timing constraint we have defined a new implementation.



	Data rate			Data rate		
	117 kbits/s			950 kbits/s		
	DSP_A	DSP_C	Time (ms)	DSP_A	DSP_C	Time (ms)
Transmitter (SW1)						
1st experience (DSPs)	96.6%	3.4%	9.99	96.6%	5.1%	10.33
2nd experience (DSP + FPGA)	11.4%	3.4%	7.96	11.4%	5.1%	8.26
Receiver (SW2)						
1st experience (DSPs)	185.5%	4.6%	19.27	185.5%	5.0%	19.33
2nd experience (DSP + FPGA)	17.1%	4.6%	9.44	17.2%	5.0%	9.49

TABLE I. Hardware component utilization rate

The results help to identify function and/or data exchanges that affect the global system performance. Changing the implementation is straightforward with the A3S tool since it just requires to modify some links (corresponding to the critical functions) and not to rebuild the whole system. Thus only two critical functions (PSH for SW1, MFL for SW2) that were previously implemented onto the DSPs are implemented onto the FPGAs (hardware solution corresponding to the 2nd experience). The remainder functions are still implemented onto the same DSPs. The new results show that for each case (SW 1, SW 2), the DSP workload was reduced (e.g. from 96% to 11% for SW 1 and from 185% to 17% for SW 2). This implementation also reduces the execution time, and the timing constraint (<10ms) is respected in each case. Thanks to the tool, the designer performs a fast analysis and is able to compare the most appropriate implementations satisfying the application and architecture constraints.

Figure 11 shows the Gantt diagram provided to the designer to analyze the resources workloads.

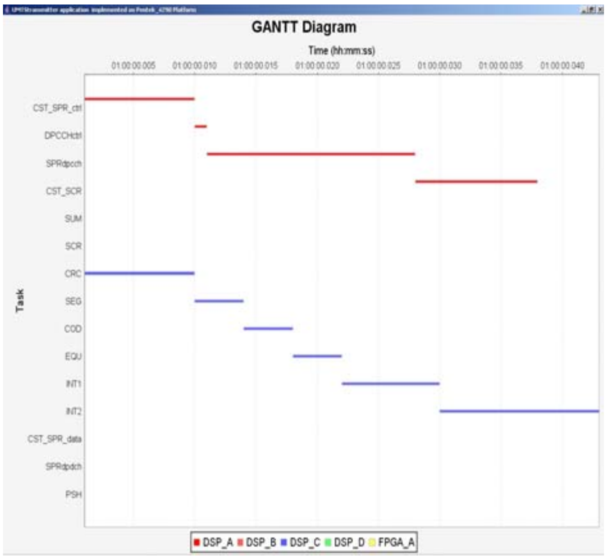


Figure 11. Gantt diagram provided to enable the analysis of the resources workloads

7. Conclusion

In this paper we have presented the UML compliant rapid prototyping A3S framework based on the UML A3S profile. It provides designers with a unified, fast and easy method to specify software applications and hardware architectures. Such an approach significantly decreases the prototyping time and enhances system reuse, which is a major concern for software radio applications. It also enables a design space exploration for rapidly testing various HW/SW mappings. Furthermore, the A3S project proposes more than a design framework. It also provides a design methodology to validate complex systems with a step by step design approach from PIM to PSM. This tool simplifies designer job by handling automatic usually heavy tasks, like coherency verifications, period task and timing computations as well as scheduling verifications. Two kinds of futur work would be necessary in order to include the A3S tool within commercial frameworks. The first on is located at the top of the design flow, it consists in a bridge with current simulation tools (SPW, matlab) from which activity diagrams could be derived. The second one deals with the link with HW and SW compilation tools, once a solution has been cleary identified, it remains necessary to automatically generate compilation and scripts for selected environments.

References

[1] J. Mitola, "The Software Radio Architecture," IEEE Communications Magazine, vol. 33, no. 5, pp. 26-38, 1995.

[2] [online] Software Defined Radio: http://www.sdrforum.org

[3] I. Bolsen et al, "Hardware/software co-design of digital telecommunication systems," Proceedings of IEEE, vol. 85, no. 3, pp. 391-418, 1997.

[4] J.P Calvez, MCSE : Spécification et conception des systèmes : une méthodologie, Masson, 1990.

[5] UML™ profile for Software Radio - OMG draft.

[6] UML™ profile for Schedulability, Performance and Time Specification – ptc/02-03-02 OMG draft.

[7] UML™ profile for QoS and Fault Tolerance Characteristics and Mechanisms- OMG revision submission.

[8] S. K. Shula et al., “High Level Modeling and Validation Methodologies for Embedded Systems: Bridging the Productivity Gap,” 16th International Conference on VLSI design, pp. 9-14, 2003.

[9] S. Edwards et al., “Design of Embedded Systems: Formal Models, Validation and Synthesis,” Proceedings of IEEE, Vol. 85, N°3, pp. 366-390, 1997.

[10] R. Gupta, G. De Micheli, ”Hardware-Software Cosynthesis for Digital Systems,” IEEE Design and Test of Computers, pp. 29-41, 1993.

[11] R. Ernst et al., ”Hardware-Software Cosynthesis for Microcontrollers,” IEEE Journal Design and Test of Computers, pp. 64-75, 1993.

[12] J. Davis et al., “Ptolemy II - Heterogeneous Concurrent Modeling and Design in JAVA,” University of California at Berkeley, September 2000.

[13] D. Araki et al., “Rapid prototyping with HW/SW codesign tool,” Proceedings. Engineering of Computer-Based Systems (ECBS), pp. 114-121, 1999.

[14] D. Gajski, "System-Level Design Methodology," ASP-DAC 2004 Pacifico Yokohama, Yokohama, Japan, January 27, 2004.

[15] [online] "Code Generation for SCA Components", white paper, http://www.zeligsoft.com/, 2005.

[16] [online] "SCA Deployment Management", White paper, http://www.zeligsoft.com/, 2005.

[17] P.Boulet, J-L.Dekeyser, C.Dumoulin and P.Marquet. "MDA for soc design, intensive signal processing experiment". FDL'03, Frankfurt, September 2003. ECSI.

[18] E.Riccobene, P. Scandurra, A. Rosti, S. Bocchio, "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC", Design, Automation & Test in Eur. Conf. (DATE), 2005.

[19] B. Steinbach, Ch. Dorotska, D. Fröhlich "Hardware Synthesis of UML-Models". Workshop on UML for System-on-Chip Design (UML-SOC'05) at Design Automation Conference (DAC'05), Anaheim, USA

[20] M.Oliveira, L.Brisolara, L.Carro, F.Wagner, "Embedded SW Design Exploration Using UML-based Estimation Tools", Workshop on UML for System-on-Chip Design (UML-SOC'05) at Design Automation Conference (DAC'05), Anaheim, USA

[21] A.Viehl, O. Bringmann, W.Rosenstiel, "Performance Analysis of Sequence Diagrams for SoC Design", Workshop on UML for System-on-Chip Design (UML-SOC'05) at Design Automation Conference (DAC'05), Anaheim, USA

[22] [online] A3S project home page: http://web.univ-ubs.fr/lester/www-lester/Projets/Codesign/A3S/English/A3Shome.htm

[23] [online] Objecteering software: http://www.objecteering.com/

[24] A. Dasdan, “Timing Analysis of Embedded Real-Time Systems,” Ph.D. dissertation, University of Illinois, 1999.

[25] Tmar H., Diguet J-P., Azzedine A., Abid M., Philippe J-L., RTDT : a Static QoS Manager, RT Scheduling, HW/SW Partitioning CAD Tool, ICM, 2004

[26] Joseph M., Pandya P., Finding response time in a real-time system, IEEE Design and Test of Computers 29 (5) (1986) 390-395.

[27] Dave P., Jha N.K., Casper: Concurrent hardware-software co-synthesis of hard real-time aperiodic specification of embedded system architectures, in: Design, Automation & Test in Europe Conf., Paris, France, 1998.

[28] Azzedine A., Diguet J-P., Philippe J-L., Large exploration for HW/SW partitioning of multirate and aperiodic real-time systems, in 10th Int. Symp. on HW/SW Codesign, Estes Park, USA, 2002.

[29] Butts J.A., Sohi G., A static power model for architects, in: 33rd ACM/IEEE Int. Symp. on Microarchitecture, 2000.

Modeling with logical time in UML for real-time embedded system design

Charles André[1], Arnaud Cuccuru[2], Robert de Simone[2],
Thierry Gautier[3], Frédéric Mallet[1], and Jean-Pierre Talpin[3]

[1] I3S CNRS/UNSA Sophia-Antipolis, France
[2] INRIA Sophia-Antipolis, France
[3] IRISA Rennes, France

Abstract. Design of real-time embedded (RTE) systems requires particular attention to the careful scheduling of application onto execution platform. Precise cycle allocation is often requested to obtain full communication and computation throughput.

Our objective is to provide a UML profile where events, actions, and objects can be annotated by "logical" clocks. Initially, clocks are not necessarily related (or even explicit). The goal of the scheduling process (and algorithms) is to regulate the data and control flows within predictable bounds. To this end it extracts clock relations that result from mapping the application onto a desired execution platform. Extra communication and buffering latencies can be introduced in the process, due to the distribution of functions onto concurrent resources. "Clocks-as-schedules" then act as activation conditions, driving these internal events and actions according to the desired activation patterns.

In the paper we describe the domain view of multiple time and logical clocks. We introduce a range of useful operations on them, and their use in various UML views. *Logical Time* is part of an on-going proposal at OMG for a profile on *Modeling and Analysis of Real-Time Embedded* systems (MARTE), that should also subsume the former SPT profile for *Schedulability, Performance, and Time*.

1 Introduction

As embedded real-time systems have become pervasive and ubiquitous in contemporary technologies, their development requires highly reliable approaches. To meet safety and performance requirements, design has to be supported by trustable mathematical basis that provides required formal concepts. A noteworthy example is the attention required to the careful scheduling of functions and operations to be performed by the application on the targeted execution platform, which demands precise cycle allocation to obtain full communication and computation throughput.

However, precise cycle allocation is error-prone and tedious. A corpus of methods has been proposed in recent years, where these precise relations are obtained by analysis and optimization techniques from more relaxed high-level modeling of systems. In other words, the final scheduling is compiled algorithmically from constraints on the application, the target HW/SW execution platform, and the allocation mapping of functions to resources. This can be done off- or on-line depending on the dynamicity and the predictability of the system being modeled.

Our current objective is to provide a UML modeling framework (profile) in which to represent the ingredients of such an approach. Indeed, UML provides broad ways to specify the different modeling views involved. But its largely "untimed" basis needs to be augmented with proper semantical annotations on temporal aspects.

While akin to many previous attempts at time modeling in UML (UML-RT [16], RT-UML [9], ACCORD/UML [10], SPT [14],. . .), our proposal still differs from them in several ways. It is based on "logical" time bases (or *clocks*) that are introduced to count/tick/trigger successive behaviors of signals, actions, objects (and so on). Clocks that are mutually independent (or only loosely coupled) provide for models of asynchronous tasks or processes. Clocks should act as generic *activation conditions*, driving the internal events and actions according to the desired timing patterns.

Precise relations between clocks can be provided by users, or infered algorithmically from the scheduling constraints described in the model. In the latter case specific optimization algorithms can take advantage from some usual limitations of RTE systems (predictability, determinism, static computation structure) to propose spatial and temporal allocations that will best fit functional applications to execution platforms. This has the effect of strenghtening the relations between clocks, constraining them to a more rigid interdependency. Ultimately all clock flows may be expressed down to a unique physical clock (but not always necessarily so). In a given methodological flow, the approach should be used to adjust the various rates involved in different parts of the application under design. Communications can be introduced by the spatial mapping of functions onto concurrent resources. Extra latencies and buffering objects may also be requested at places to regulate the data and control flows within predictable bounds.

Logical clocks associated with successive object or action behaviors are a convenient way to represent explicit schedules as first-class citizens of the model. Such schedules can then be named, visualized, and computed upon. Results can be displayed back to the designer as a clock schemes refinement. It provides effective information on the mapping decisions taken towards implementation.

We describe clock connectors, used to combine and compare them. Examples are over- or down sampling, delay and resynchronizing operators, and so on. Sometimes the connectors do not provide a unique transformation on tehir input clocks, but rather introduce new constraints that allow a range of solutions (a given schedule is then obtained by *solving* the set of aggregated solutions by a scheduling technique). Clocks can be hierarchically organized, fully or partially. A clock can be a periodic subclock of another (amongst other). Ultimately, a clock that upsamples all others in the system can be thought of as "discrete physical time", but the existence of a link to physical time is not mandatory in our modeling framework.

The classical execution semantics of UML is asynchronous/untimed, which is just fair as long as the timing constraint information is not specified in the model. In our case an extended semantics should comply with the scheduling directives induced by the logical clocks and their constraints.

The paper is organized as follows: section 2 describes the time model and its logical clocks, their insertion in the application model and their allocation to an execution platform model. Section 3 describes a case study that highlights the approach. We conclude with perspectives.

Related works Our approach should appeal to readers familiar with model-driven design based on mathematical *Models of Computation* [4, 11]. The notion of logical clocks providing the schedule framework owes to the theory of tagged systems [13]. *Synchronous reactive formalisms* [3] are examples of languages that handle (logical) time as explicit mechanisms at the very heart of their semantic foundations. Schedule computations from data-flow based application descriptions were developed in the theory of *Timed Event Graphs* [5, 2]. It has found many usages, with some important developments for software pipelining [8] and hardware circuit timing [15]. It found some early incarnation in the context of synchronous programming with the theory of affine clocks [18]. The explicit representation of schedules in the discrete/periodic/regular case was introduced in the theory of *N-synchronous* processes [6].

2 Modeling framework

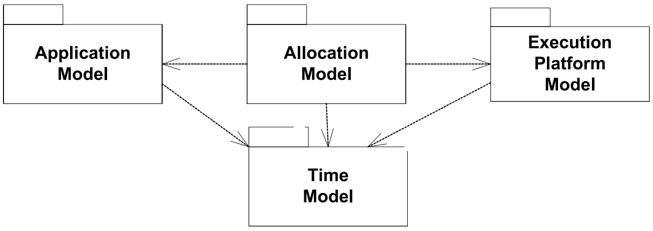


Fig. 1. The various models and their package architecture

Our approach relies on the modeling framework displayed in figure 1. A *functional application* is to be *allocated* onto a candidate architectural *execution platform*. The application may exhibit potential concurrency and relative asynchrony between various treatments, as well as subsystems running at various related speeds. The platform may consist of mixed hardware and basic software parts. The *allocation mapping* consists of both *spatial distribution* and *temporal scheduling* of functional operations onto platform resources and services. Note that the words “functional” and “architectural” here do not exactly match “behavioral” and “structural” notions. Functional applications have a strong behavioral impact, but contain also hierarchical structure descriptions; execution platforms, while providing the architectural block-diagram of resources and connections, also describe some basic behavioral “service” aspects of the platform.

The allocation mapping between the application and the platform can be entirely specified by the designer, or computed by analysis from a number of characterization figures. It will result in a refinement of the time model of the application by the architecture constraints. This time refinement will be reflected in a tighter set of relations between logical clock schedules. For instance relatively independent clocks can be interleaved when functions are mapped to the same resource. The importance of schedule descriptions as explicit model elements should fully appear here.

2.1 Time Model

Our objective here is to introduce conceptual definitions related to our vision of *Time*. The essential ones are represented in their domain view in figure 2.

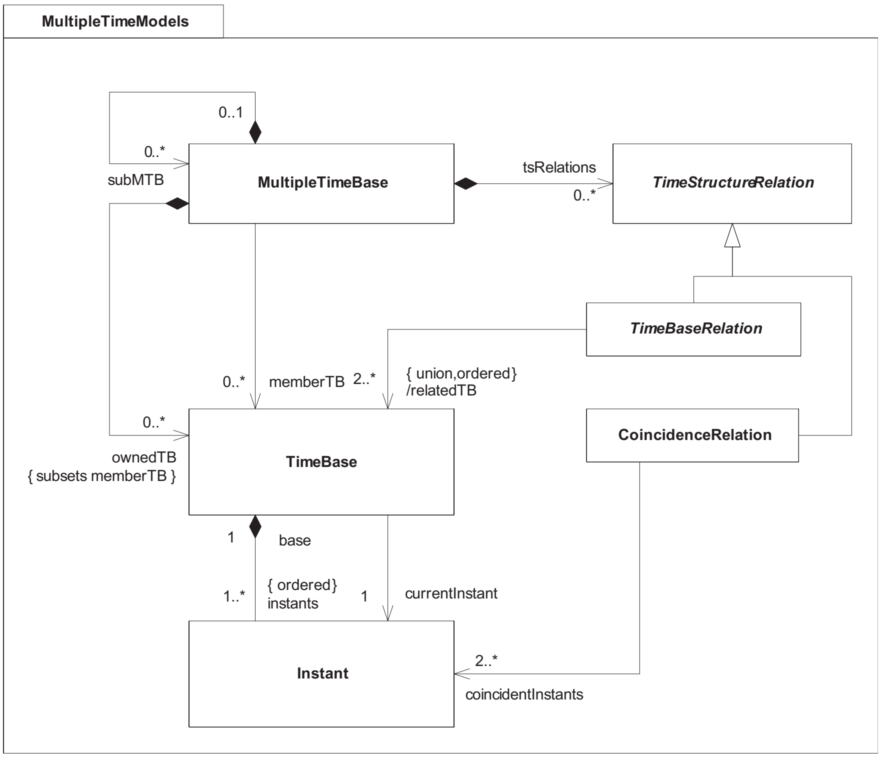


Fig. 2. The time base model

The MultipleTimeModels package (Fig. 2) introduces a structural view of Time. Time can be *simple* (totally ordered), or *multiple* (partially ordered in the form of several, loosely coupled time bases). Precedence of instants can thus be defined as a partial relation.

A clock attaches quantitative information (time values) to instants or set of instants of time bases. A clock can be either *logical* or *chronometric* (the latter indicating that time is supposed to be measured from physical devices, external to the model by definition). In the paper we shall concentrate on (more novel) logical time.

Even though time base and clock are two different concepts, in this paper, the word clock will often be used in place of time base, this in accordance with standard usages.

Time can be *discrete* or *dense*. We shall stick here to discrete time, where Time Bases can be seen as generated from clock ticking events. When modeling repetitive tasks with discrete time it becomes possible to express explicit schedules relating the respective paces of clocks rather easily (which does not mean that it is always impossible in the dense case).

Establishing tighter relations between clocks shall be the main scheduling purpose in the methodological system design activity associated with our modeling approach. Coincidence of instants ticked by distinct clocks is allowed (simultaneity), so that synchronous activities can be modeled.

Coincidence between instants of different clocks are expressed by *time structure relations*. These relations are special UML constraints specified by predicates on clocks. There are three main kinds of time structure relations: interleaving, decimation, and rooted relations. *Interleaving* merges clocks while preserving each clock instant ordering. *Decimation* extracts subclocks: a clock A is *finer than* a clock B when A ticks at least each time B does. *Rooted relations* combine clocks that share a common finer clock called their root.

A non-exhaustive list of useful clock predicates can be provided (remember that these predicates are often relational in the sense that they do not lead to a unique solution, and that a specific solution is provided only when a specific deterministic scheduling is given). *Equal(H,H')* states that two clocks are identical; conversely *disjoint(H,H')* states that they share no instant. *Shift(H,n)* starts with the nth tick of H. *Coarser(H,H')* states that H' is a subclock of H; *Finer(H',H)* stands for *Coarser(H,H')*. *Faster(H,H')* states that the nth tick of H occurs before the nth tick of H' ; *Slower(H',H)* stands for *Faster(H,H')*. *Subsample(H,H',Pattern)* state that H' ticks only on those instants of H selected by the pattern (usual patterns are periodic words on $\{0,1\}$ or affine expressions $ax+b$); *Oversample(H',H,Pattern)* stands for *Subsample(H,H',Pattern)*. *SameRate(H,H')* states that the two clocks have asymptotically the same rate. Furthermore, *MaxDrift(H,H',d)* means that the difference between the two clocks is bounded by d ticks. *SampleTo(H,H')* projects each tick of H onto the tick of H' that immediately follows it; it is ill-defined if there are two successive ticks of H without a tick of H' in between, possibly coincident with the first of the two ticks of H.

In the example of section 3 we shall use the decimation relation $B = Periodic\text{-}Decimation(A, 10, 0)$ stating that clock A is finer than clock B, and that each instant of B coincides with every 10 instants of A (10 is the period, 0 is the offset). More generally, a decimation is characterized by a strictly increasing sequence of integers (indexes of coincident instants), or even more concretely by a sequence of bits, called a *filter pattern*. If A is finer than B, then a '1' at a given index position i in the sequence indicates that clock B ticks at the i^{th} tick of A.

Formal definitions for clock predicates are provided in a research report [1].

2.2 Application Modeling

Applications should be represented using the familiar UML modeling views for structural and behavioral aspects: state, activity and interaction diagrams, structured classes and components. In the field of RTE systems it is often the case that activity and "component blocks" diagrams play a particular role, as noticed through their emphasis in SysML for instance.

To build our application model, the primary extension to these models shall be the introduction of the logical clocking information available (or requested) as annotations.

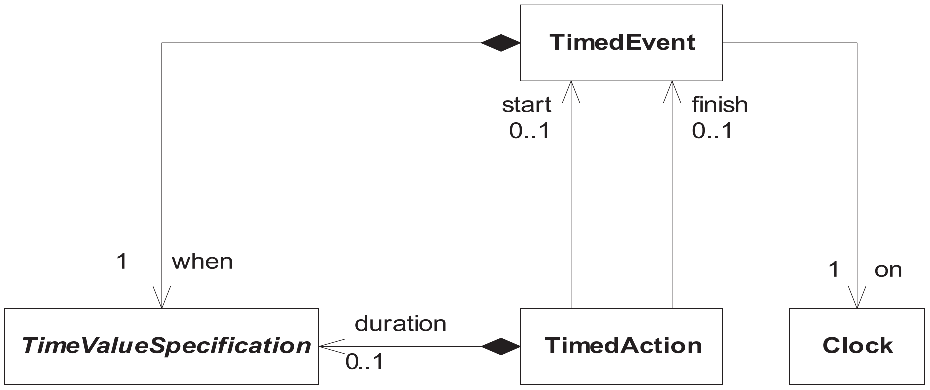


Fig. 3. Time entities

The objective is to provide schedules for elementary behavioral elements such as *actions* and *events*. Timed behaviors are tagged with two events denoting the start and the finish of this behavior. Events, when timed, are annotated with a time value that refers to a clock. The activation instants of behaviors on a given clock can be provided as *absolute values*, but much more often as *relative values* introducing latencies and durations. Such values are provided usually as TimeValueExpressions. For instance, if the start event of a given activity is annotated with the expression "each tick on clock1", it means that this activity is triggered by the clock "clock1". Hence, activities are synchronous if there are all triggered by the same clock. When all actions within an activity refer to synchronous activities (Call Behavior Action) the activity is stereotyped "sActivity" for synchronous Activity. The extension of relevant modeling elements to their `Timed` versions is displayed in figure 3.

Timed semantics The operational semantics of UML is basically untimed (some would call it *asynchronous*). When timing considerations are introduced, they are often added as external "non-functional" constraints, and their satisfaction is not demanded explicitly in the "official semantics". This is certainly fine to define a computational model that works when these timing annotations are not provided, and we certainly do not want to question this model in such case. But when explicit timing (logical or physical) is specified in the model, the semantics should of course take it into account (or else the UML semantics be discarded for modeling intentions that would only rely in its diagrammatic views and discard its meaning). Following Bran Selic [17], we see two places where this is important:

inter-object communications. Example questions here are: What is the time relation between the *send* and *receive* actions on a signal event? How could one model rendez-vous synchronization when modeling demands? The actual arriving time can have a drastic impact on real-time scheduling techniques (and then on the ordering of behaviors in a faithful operational semantics).

intra-object communications Actions are triggered by object/data flows and control flows. But in the current state of UML 2.0 there is not much differences betwen the two kinds of flows. *Data-flow* models are found very useful for abstract modeling, but hardly used for real-life semantics. Instead they are usually transformed (by classical scheduling) in a model where all information on cycle time activation is carried by the control flow (it "provides" control), and the data-flow is weakened to a *data-path*; indeed, control-flow design is supposed to ensure that the proper data have arrived in the proper locations (here, at the input pins of the action), when it is activated.

2.3 Execution Platform Modeling

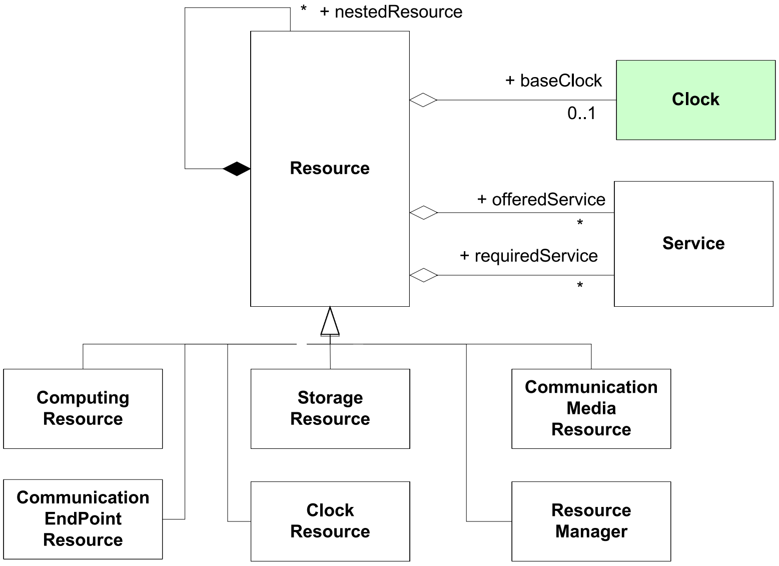


Fig. 4. Execution platform metamodel, with examples of hardware specific resources refinement (simplified diagram)

The purpose of the Execution Platform Model is to enable embedded systems designers to specify and dimension the architectures meant to support the applications. The actual allocation (of application to platform) can be specified entirely by the designer, or sometimes, computed by analysis tools [12]. Such tools require basic information on the computation and communication costs for basic functions, from which they attempt to minimize the overall cost of well-chosen allocations. Algorithmic details are out of the scope of this paper. The result can be displayed as tighter clock relations. Typically a new ground clock is introduced, on which existing clocks are synchrononized to the desired effect. For instance functions that share a resource must be interleaved exclusively.

As illustrated in Fig. 4, the model mainly introduces two concepts: *resources* and *services*. Structurally, an execution platform is a block-diagram of resources of several natures: *computing, storage, communication endpoints* (performing send/receive actions), and *interconnect media*. Platform services use a predefined number of resources in a way described as an interaction scenario. *Generic* services describe natural aspects of the platform and can be used to introduce *costs* at a higher description level. *Specific* services can be used to provide descriptions that come to the level of application functions. This is used to bridge the possible gap in atomicity level to group the necessary platform behaviors and resources so that it can realize the function directly.

No assumptions are made on the granularity of the resources considered in the specification of an execution platform. According to nature of the application, an execution platform can be a coarse grain description of a basic software operating system offering services for thread and memory management, or very fine grain view point of a hardware execution platform (down to ASICs). Note that the hierarchical aspect of the metamodel (composition relation between resources in Fig. 4) enables all kinds of layering specifications. For example, the bottom layer of an execution platform (typically specified as an assembly of fine grain hardware resources) can be abstracted by a higher level software execution layer.

Resources timing and clocking information are often of a more physical nature as those of the application. Time bases provide a specification of computing speeds for hardware elements. Durations provide an account of computational complexity. For example, in a hardware execution platform, physical relations typically exist between the operating frequencies (clocks) of the various resources (e.g., a bus clock ticking one time every ten ticks of the processor clock, i.e., the bus clock is ten times slower than the processor clock but is synchronized with it).

2.4 Allocation modeling

The *Allocation* metamodel deals with the mapping of functional *Application* elements onto*Execution Platform* resources and services. The same concern exist in SysML (under the same name), and in SPT (where it was called *Realization*). It is usually considered in connection with *Abstraction/Refinement* mechanisms. But while allocation provides an "horizontal" mapping between previously independently described entities, abstraction/refinement provides "vertical" association notions *inside* each respective representation (Application or Execution Platform), respecting their global structure while allowing to lift it up or down to match the level of details of the other side.

With potentially clocked systems on each side, an important concern of the allocation process will be to associate the (more logical) clocks of the application to the (more physical) clocks of the execution platform. In the simple case of untimed application, it will mainly consist in assigning a time cost for executing that function on that resource (or for executing that communication on that interconnect medium). More generally, we want to provide contraint laws amongst clocks. Here again SysML "parametrics" (or "contraints") models could be used. Note that, while Allocation and Constraint models are rather independent in SysML, here the temporal laws are an integral part of the Timed Allocation process.

As already mentioned, a more accurate and tighter application schedule is usually meant to be computed as a result of Timed Allocation, using the time constraints from the execution platform to refine the time relations of the application. Ultimately, this might assign physical clock cycles to the execution of the elementary actions. This is shown in the example of section 3. Note that in case of code parallelization/vectorization, or conversely of timesharing between distinct tasks, the clock scheme of the application can get distorded by these additional constraints imposed by the mapping.

The Allocation metamodel offers different kinds of relationships (structure to structure, behavior to structure, behavior to behavior), that will not be described in details in this paper. But ultimately, an action considered as atomic in the application must be related (directly or indirectly through an analysis process) to the (possibly multiple) execution of one or more services supported by one or several resources of the targeted execution platform. This property is illustrated in Fig. 5, where Collaboration and Behavior concepts have a similar semantics to homonymous UML 2 concepts (respectively the set of structural elements, i.e., resources, that will collaborate to "realize" the action, and the description of the interactions, i.e., service executions, between the different parts of the collaboration).

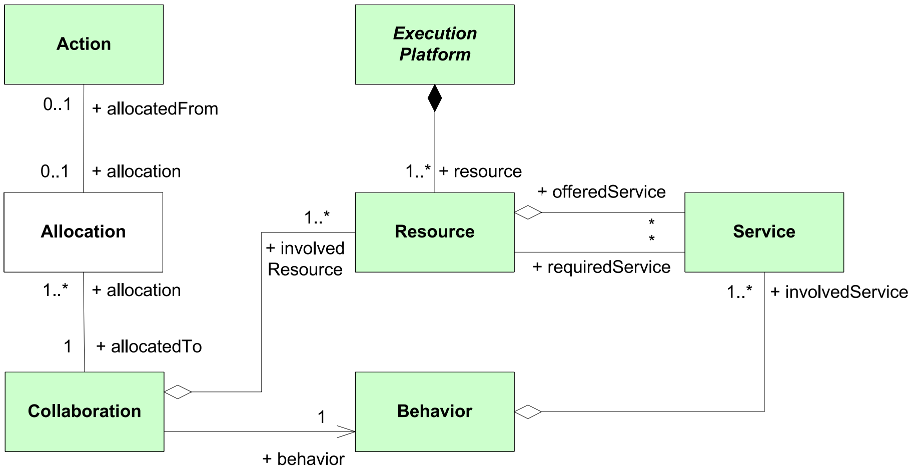


Fig. 5. Allocation metamodel (partial)

3 An illustrative example

As an illustrative example, we consider the downscaling of a high definition (HD) video image into a standard definition (SD) image. This example is primarily meant to show the use of *Clocks-as-schedules*. In that sense, it hardly uses Execution Platform and Allocation modeling, and focuses on Timed models inside application instead.

The downscaling application has been modeled with Multi-Periodic Process Networks [7], a model influenced by Petri nets, data-flow graphs, and Kahn Process Networks. A HD image consists of 1080 lines, each made of 1920 pixels. A SD image has 720 lines with 480 pixels each. The transformation reduces the number of pixels per line (ratio of 8:3), and the number of lines (ratio of 9:4). So, altogether the output pixel rate is $\frac{3}{8} \times \frac{4}{9} = \frac{1}{6}$ of the input rate. Functionally, the transformation can be decomposed in the horizontal filtering of each HD lines, followed by the vertical filtering of the resulting lines. Filtering includes smoothing (each new pixel results from a weighted sum of a neighborhood) and a decimation (discarding pixels). This transformation must be done in real-time: the pixels of input HD image are received at a rate imposed by the inClk clock, and the pixels of the output SD image have to be delivered at a rate imposed by the outClk clock. The two clock frequencies are specified in the standards. At the implementation level, horizontal and vertical filterings are performed in a pipeline mode, which calls for a precise schedule of elementary operations. For simplicity we describe only the horizontal filtering.

Given the specification of the application and a target execution platform, we proceed as follows:

1. A static model of the main data structures handled by the application is designed. It brings out dimensional data-flow aspects, closely related to repetitive processings.
2. The functionality of the application is expressed by activity diagrams or pseudo algorithmic representations. Local logical clocks should be introduced at this step to support scheduling.
3. Activities are allocated to services supported by an Execution Platform. Information about the duration of the service executions is collected from the platform model and integrated in the logical clocks.
4. A clock calculus determines the schedule of all actions. If the external timing constraints are not satisfied, then the activities or the local clocks have to be modified, and the procedure is applied again from step 3.

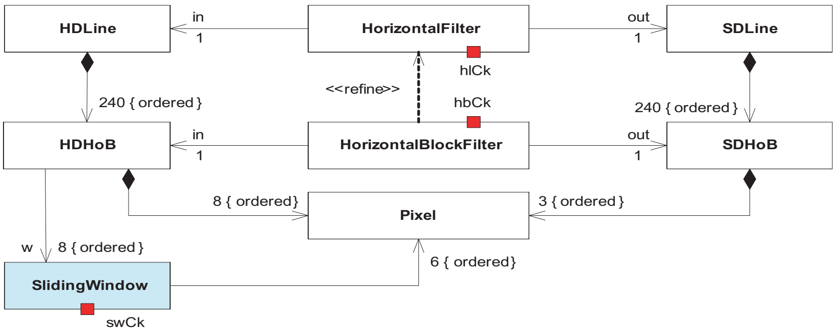


Fig. 6. Static model

Step 1 The static model is given in Fig. 6. A HD line consists of 240 HD Horizontal Blocks (HDHoB), each made of 8 pixels. On the SD side, a line is also made of 240 SD Horizontal Blocks (SDHoB) of 3 pixels each. The filtering of a HD line can be refined into 240 horizontal block filterings. The horizontal block filtering reduces the line length in a ratio of 8:3.

Step 2 The logical clocks introduced for the HD line filtering are justified below and their equations are gathered in Table 1.

#	clock name	equation
1	pxInClk	$\mathrm{alternation}(\mathsf{inClk}, \mathsf{pxInClk})$
2	HDHoBClk	$\mathsf{HDHoBClk} = \mathsf{pxInClk}\vert_{(1.0^7)^*}$
3	HDLineClk	$\mathsf{HDLineClk} = \mathsf{HDHoBClk}\vert_{(1.0^{239})^*}$
4	smoothClk	$\mathsf{smoothClk} = \mathsf{HDHoBClk}\vert_{(10100100)^*}$

Table 1. Clocks after step 2

The pxInClk clock samples the pixels received at the rate of inClk without loss or repetition. This holds whenever ticks of the two clocks alternate. Alternation is a special interleaving relation.

Since a HD horizontal block consists of 8 received pixels, the clock attached to the start of a HDHoB filtering (HDHoBClk) must be 8 times slower than pxInClk. This is expressed by Eq.2 using the *clock downsampling* relation, which is a decimation relation with a periodic filter pattern, $(1.0^7)^*$ in this case.

Since a HD line contains 240 blocks, the clock attached to the start of a HDLine filtering (HDLineClk) must be 240 times slower than HDHoBClk.

A block filtering consist of 8 smoothing operations on a window. But because of the pixel decimation, only 3 out of the 8 are necessary. So, the 8 iterations making a block filtering are not identical. The body of the loop does or does not compute a smoothing function and generate a new pixel. This kind of variable iterations are elegantly expressed by a clock expression. Let smoothClk be the clock starting a body execution with smoothing and pixel generation. The filter pattern 10100100 preserves 3 results out of 8 computations. It reflects a design decision that fixes which pixels must be discarded, in an evenly spread way.

The WindowFiltering activity triggered by smoothClk is

1. Get the incoming pixel.
2. Push it in the (sliding) window w (the oldest pixel is lost).
3. Compute a dot product: $s = \sum_{k=0}^{k=5} w[k] * a[k]$, where $w[k]$ is the value of the k^{th} pixel in the window, and a is an array of weighting coefficients.
4. Create a pixel with value s and insert it in the SDHoB.

The complementary clock of smoothClk (i.e., $\mathsf{HDHoBClk}\vert_{(01011011)^*}$) triggers only the first two actions of the above activity.

The behavioral model of the application is specified by activity diagrams (Fig.7). Standard UML activity diagrams are extended to support the concept of clocks. We have introduced special pins, called *clockPins* used to trigger actions, and a new compartment for clock relations. Note that some clocks (e.g., rCk) are local to an activity.

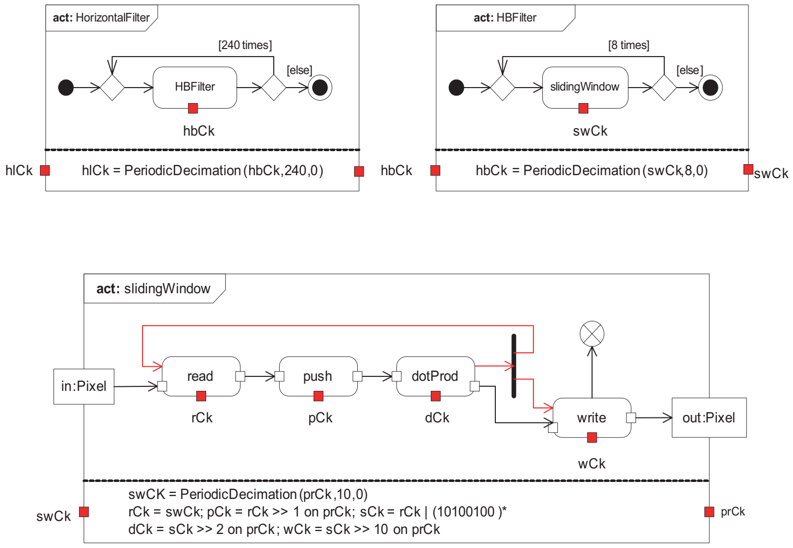

act: HorizontalFilter
[240 times]
[else]
HBFilter
hbCk
hlCk
hlCk = PeriodicDecimation (hbCk,240,0)
act: HBFilter
[8 times]
[else]
slidingWindow
swCk
hbCk
hbCk = PeriodicDecimation (swCk,8,0)
swCk
act: slidingWindow
in:Pixel
read
push
dotProd
write
out:Pixel
rCk
pCk
dCk
wCk
swCk
swCK = PeriodicDecimation (prCk,10,0)
rCk = swCk; pCk = rCk >> 1 on prCk; sCk = rCk | (10100100)*
dCk = sCk >> 2 on prCk; wCk = sCk >> 10 on prCk
prCk

Fig. 7. Behavioral model

Step 3 In order to select an Execution Platform, we have to consider required performance. 25 full images are sent every second. Each image has 1125 lines (1080 visible lines + 45 service lines in the HDTV NHK standard). Each line has 1920 pixels. Thus, the inClk has a frequency of $25 \times 1125 \times 1920 = 54.10^6$ Hz, hence a period of 18.5 ns. So small an amount of time forbids the use of general purpose execution platform. An ASIC solution is chosen. Coefficients for the dot product are negative power of 2 so that the dot product consists of simple bit-shift followed by additions.

The resources are 2 registers, 1 shift-register, and a tailored ALU. Tab. 2 contains pertinent information about the execution platform. prClk is the clock of the hardware. Duration is expressed in number of cycles of this clock.

From Table 2 we extract new clock relations gathered in Table 3. The duration for the execution of the WindowFiltering activity is $1 + 1 + 8 = 10$ cycles of clock prClk (note that the write action is performed concurrently with the next read action). So, this clock should be 10 times faster than clock pxInClk (Eq. 5). In this case, we use a *clock upsampling relation*, a special decimation relation. On the other hand, the sliding window has to be filled-in before it can operate at its full speed. This needs a delay of 5 instants on the HDHoBClk clock (Eq. 6). The *delay relation* denoted by $>>$ is another instance of decimation relation, characterized by the filter pattern $0^{\,d}.1^{*}$ for a delay of

Resource	Type	Service	Duration	Clock
pxInBuf	Register	get():Pixel	1	prClk
slidingWindow	Shift-Register	push(p:Pixel)	1	prClk
		get(i:integer): Pixel	1	prClk
filter	ALU	dotProduct():Pixel	8	prClk
pxOutBuf	Register	put(p:Pixel)	1	prClk

Table 2. Execution Platform

d. Now, a fine schedule of actions can be derived from the WindowFiltering activity specification and the known durations of the actions. For instance, the get action on pxInBuf has to be scheduled one instant of prClk after the sampling of a pixel. This is expressed by Eq. 7. The push action in the sliding window has to be scheduled two instants after the sampling of a pixel (Eq. 8), and so on.

#	clock name	equation
5	prClk	$\mathsf{prClk} = \mathsf{pxInClk}\|^{(1.0^9)^*}$
6	$\mathsf{HDHoBClk}'$	$\mathsf{HDHoBClk}' = \mathsf{HDHoBClk} \gg 5$
7	pxInBufGetClk	$\mathsf{pxInBufGetClk} = (\mathsf{prClk} \gg 1)\|_{(1.0^9)^*}$
8	slidingWindowPushClk	$\mathsf{slidingWindowPushClk} = (\mathsf{prClk} \gg 2)\|_{(1.0^9)^*}$

Table 3. Clocks after step 3

4 Conclusions and Future Directions

We have provided a modeling framework to represent time schedule informations in Real-Time Embedded applications. The approach is model-based and relies fully on existing UML modeling paradigms. Explicit schedules are represented as logical clocks and clock relations. In the case of predictible periodic behaviors they can be computed upon, but the modeling scope is not limited to this case; in the larger spectrum they can be kept as relations. We focused on discrete clocks, but rational clocks could be defined as well.

The approach relies on time refinement, by which a set of loosely related clocks can be inter-scheduled to one that encompasses them all. Execution platform models provide mandatory duration values for computations, and allocation mappings associate these constraints to the application demands. Defining ad-hoc platform *services* brings the platflorm closer to the application.

Further work should be conducted to demonstrate the use of explicit logical clock scheduling in broader scope. Real case studies on execution platform models would also lead to a better understanding of the approach. We are currently implementing these concepts into UML profile modelers, and presenting them at the OMG in the framework of the MARTE profile proposal (*Modeling and Analysis of Real-Time Embedded systems*).

Our graphical representations should be refined so as time schedules and clock relations can be precisely represented on UML diagrams in the most practical way.

References

1. Charles André, Arnaud Cuccuru, Robert de Simone, and Jean-Pierre Talpin. Modeling with logical time in uml for real-time embedded system design. Technical report, Sophia Antipolis, 2006. Rapport de Recherche INRIA/RR–5895.
2. F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.
3. Benveniste, Caspi, Edwards, Hallbwachs, Le Guernic, and de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1), 2003.
4. J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, April 1994.
5. J. Carlier Ph. Chrétienne. *Problème d'ordonnancement: modélisation, complexité, algorithmes*. Masson, Paris, 1988.
6. Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks. In *POPL 2006 Proceedings*, January 2006.
7. Albert Cohen, Daniela Genius, Abdesselem Kortebi, Zbigniew Chamski, Marc Duranton, and Paul Feautrier. Multi-periodic process networks: Prototyping and verifying stream-processing systems. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 137–146, London, UK, 2002. Springer-Verlag.
8. Vincent H. Van Dongen, Guang R. Gao, and Qi Ning. A polynomial time method for optimal software pipelining. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing*, volume 634 of *LNCS*, pages p613–624, 1992.
9. Bruce Powell Douglass. *Doing Hard Time: Developping Real Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
10. S. Gérard, F. Terrier, and Y. Tanguy. Using the model paradigm for real-time systems development: Accord/uml. In *OOIS'02-MDSD*, volume 2426 of *LNCS*, Montpellier (F), 2002. Springer-Verlag.
11. Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufman, 2003.
12. C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991.
13. E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
14. OMG. *UML Profile for Schedulability, Performance, and Time Specification*. Object Management Group, Inc., 492 Old Connecticut Path, Framing-ham, MA 01701., January 2005. OMG document number: formal/05-01-02 (v1.1).
15. François R.Boyer, El Mostapha Aboulhamid, Yvon Savaria, and Michel Boyer. Optimal design of synchronous circuits using software pipelining. In *Proceedings of the ICCD'98*, 1998.
16. Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real Time Systems. Technical report, ObjecTime, 1998. http://www.objectime.com.
17. Bran Selic. On the semantic foundations of standard uml 2.0. In *SFM-RT 2004*, volume 3185 of *LNCS*, pages 181–199. Springer-Verlag, 2004.
18. Irina Smarandache. Transformations affines d'horloges: application au codesign de systèmes temps-réel en utilisant les langages signal et alpha. In *Thèse de l'Université de Rennes*, 1998.

Analysis and Modeling of Real-Time Systems with Mechatronic UML taking Clock Drift into Account[⋆]

Holger Giese, Stefan Henkler, and Martin Hirsch[⋆⋆]

Software Engineering Group,
University of Paderborn,
Warburger Str. 100,
D-33098 Paderborn,
Germany
[hg|shenkler|mahirsch]@uni-paderborn.de

Abstract. A number of approaches for the modeling and automatic verification of UML models of real-time systems exists. These approaches make the strong assumption that the clocks employed in the models are perfect. In practice, however, clocks on different nodes usually show different values and speed. In real-time systems, the related clock characteristics of clock drift, rate, offset, and precision must thus be considered during modeling and verification. In this paper, we present an approach based on our MECHATRONIC UML approach which at first captures information about clock drift and precision at the platform independent level, in the platform model, and at the platform specific level. Secondly, we present an emulation scheme which considers the effects of clock drift and permits the compositional verification of ATCTL properties and deadlock freedom for a known precision.

1 Introduction

A number of approaches for the modeling and automatic verification of UML models of real-time systems exists [1, 2]. These approaches make the strong assumption that the clocks employed in the models are perfect. In practice, however, clocks on different nodes usually show slightly different behavior due to clock drift (cf. [3, 4]).

The difference between clocks can be restricted to very low values in tightly coupled networks, but may also be considerably large or even unbounded in the long run for independently operating subsystems. In most cases, at least the precision (the maximal offset between different clocks) is known. The FlexRay bus system, for example, guarantees a precision of $3\mu s$ [5].

While approaches exists which use Timed Automata verification to verify a certain precision for a given clock synchronization protocol [6] or which take clock drift into account by adjusting the model manually in a problem specific manner [7], to the best

[⋆] This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

[⋆⋆] Supported by the University of Paderborn.

of our knowledge no general applicable approach which can thus be automated has been proposed yet.

Even in the case that the specific clock synchronization protocol is known and can be modelled with Timed Automata (cf. [6]), it seems that to encode the application specific time constraints on top of the explicit modeled time, Timed Automata with additive clock constraints are required. However, Timed Automata with addition of clock values in the time constraints cannot be checked as their emptiness problem is undecidable (cf. [8]). Therefore, we propose to look for a pessimistic emulation of the timed behavior with clock drift rather than a precise one, as otherwise automatic verification seems not feasible.

In this paper, we present an approach which, at first, captures information about clock drift and precision at the model level and extends our MECHATRONIC UML approach [9–11] for the compositional modeling and verification of real-time systems with embedded control functionality. Secondly, we present an emulation scheme which pessimistically considers the effects of clock drift and therefore permits to compositionally verify ATCTL properties and deadlock freedom for the system, taking the effects of clock drift into account.

The paper is structured as follows: We first precisely define the problem of clock differences and clock drift in Section 3. Then, we outline in Section 4 how the required or known bounds on clock differences and clock drift have to be taken into account in platform independent and platform dependent UML models. In Section 5, the underlying techniques for the verification of UML models with real-time and their extension for the case of clock differences and clock drifts is outlined. The validity of the chosen verification scheme is demonstrated in Section 6 by showing that the emulation is a pessimistic abstraction and thus ATCTL formulas and deadlock freedom checked for the emulation can also be guaranteed for the real-time behavior of the system with the known clock difference and clock drift. The paper closes with a discussion of related work in Section 7, a conclusion, and an outlook on planned future work.

2 Example

To outline our approach, we employ as running example the air gap optimization system for a linear drive, which has been developed within the collaborative research center 614[1] of the German National Science Foundation (DFG) and applied concretely to the RailCab research project.[2] The RailCab project uses a passive track system with shuttles that operate individually. The shuttles use a linear drive actuation techniques similar to the Transrapid.[3] In contrast to the Transrapid, the existing railway tracks will be reused and just need to be extended by adding a stator in the middle of the tracks. The counterpart, the rotor, is in the shuttle itself. The infrastructure of the system is based on a satellite supported positioning network and wireless communication network for the communication between shuttles and stationary installations.

[1] http://www.sfb614.de

[2] http://www-nbp.upb.de/en/index.html

[3] http://www.transrapid.de/en/index.html

The idea of the air gap optimization system is to minimize the air gap between a shuttle and a track to enable the optimization of the energy consumption. As shown in Figure 1, the distance between the rotor and stator needs to be minimized to reduce the energy needed for transmitting the power through the magnetic field. But if the distance is too close, the gravity between rotor and stator will be too strong and hence the rotor and stator are pressed together. Thus the control becomes a critical aspect and has to be addressed in an appropriate manner (e.g. fail-safe behavior). Our proposed modeling approach will address this aspect. To enable the adaption of the distance between a track and a shuttle, the two rotors of a shuttle can be manipulated in their height. As the track changes permanently due to abrasion and tiny movements of the track foundation as well as temporarily ones due to the weather conditions, the a priori regulation of the distance between rotor and stator by considering the optimization of the energy would require a large distance and thus would be very inefficient.

Therefore, the distance must be regulated dynamically. The first idea is to use sensors to measure the distance between rotor and stator. Based on these measurements, the air gap can be regulated. As the measurements could not be directly, in zero time, used for the control of the air gap, the data of the measurements are recorded and communicated to a control point, which has still enough time for using the data for regulation. Considering Figure 1, the measured air gap at a specific point at the track of the front of the leading shuttle can be communicated to the back of the shuttle itself and, perhaps, can be used to adapt the air gap based on the measured data at the front. Furthermore, the data can be communicated to the last shuttle and used for optimization of the air gap.

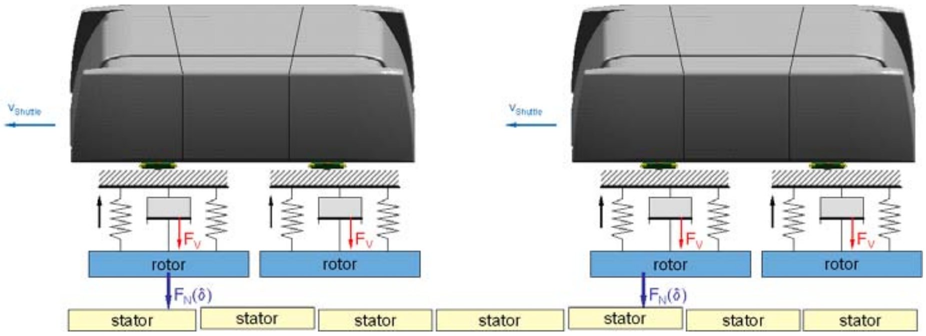


Fig. 1. The air gap module

In the next section, we review some fundamental definitions and discuss the parameters that characterize the behavior and the quality of clocks in distributed systems.

3 Foundations

In principle, each component of a system has access to information about the time through a clock implemented on the system. However, there is an unavoidable drift of local clocks, i.e. a local clock always proceeds faster or slower than the real time. Considering a number of cooperating components, this implies that in the course of time, their respective local clocks will deviate more and more from the real-time and from each other. Thus, as long as no correction action is performed, clocks on different nodes tend to drift apart. In this section, the parameters that characterize the behavior and the quality of clocks are discussed in accordance with [3].

Definition 1. *We have to distinguish between* physical clocks (pc_i) *and a* reference clock (z). *With t we denote a point of real time and with $pc_i(t)$ the value of the physical clock pc_i on time t (the same holds for the reference clock z). In more detail a physical clocks runs locally on a system component. In contrast to the physical clocks, the* reference clock *is in perfect agreement with the international standard of time, e.g. the International Atomic Time (TAI).*

Definition 2. *The* clock drift *of a physical clock pc_k between two points t and t' of real time where $t' > t$ is the frequency ratio between clock pc_k and the reference clock z at point of time t:*

$$\mathit{drift}^k_{t,t'} = \frac{pc_k(t') - pc_k(t)}{z(t') - z(t)}.$$

Definition 3. *For describing the effects of the clock drift, we introduce the term* drift rate, *which gives a value showing how the physical clock behaves with respect to real time:*

$$\rho^k_{t,t'} = \left| \frac{pc_k(t') - pc_k(t)}{z(t') - z(t)} - 1 \right|.$$

A perfect clock will have a drift rate of 0, hence the clock drift will be 1.

Definition 4. *The* maximum drift rate ρ^k_{max} *of a clocks k is defined as the maximum drift rate which can be observed*

$$\rho^k_{max} = \max_{\forall t < t'} \{\rho^k_{t,t'}\}.$$

Typical maximum drift rates ρ^k_{max} are in the range of 10^{-2} to 10^{-7} sec/sec, or better.

Another parameter to characterize the behavior of two clocks is the *offset*.

Definition 5. *The* offset *at a point of time t between two clocks pc_j and pc_k is defined as*

$$\mathit{offset}^{j,k}_t = |pc_j(t) - pc_k(t)|.$$

Definition 6. *The* precision *of a set of clocks $\{pc_1 \ldots pc_n\}$ is defined as the maximum offset between any two clocks*

$$\Pi = \max_{\forall 1 \leq j,k \leq n, t} \{\mathit{offset}^{j,k}_t\}$$

To stress that the *drift* can be bounded by the difference of two clock values, we can conclude from Definition 2, Definition 3, Definition 5, and Definition 6 that follows

$$\rho^k_{t,t'} = \left| \frac{pc_k(t') - pc_k(t)}{z(t') - z(t)} - 1 \right| \overset{t^z := z(t') - z(t), t^z > 0}{=} \frac{|(pc_k(t') - pc_k(t)) - t^z|}{t^z}$$

$$\Rightarrow t^z\ \rho^k_{t,t'} = |(pc_k(t') - pc_k(t)) - t^z| \wedge t^z(1 + \rho^k_{t,t'}) = |(pc_k(t') - pc_k(t))|$$

$$\Rightarrow \begin{cases} t^z(1 + \rho^k_{t,t'}) = (pc_k(t') - pc_k(t)) & , (pc_k(t') - pc_k(t)) \geq t^z \\ t^z(1 - \rho^k_{t,t'}) = (pc_k(t') - pc_k(t)) & , (pc_k(t') - pc_k(t)) < t^z \end{cases} \tag{1}$$

We first, for simplification, substitute the denominator which is the difference between two points of time t and t' of the reference clock by the constant t^z and assume, that the difference is always positive. From this equation we can conclude, that the clock rate, multiplied with the differenz t^z, is still bounded by the absoulte value of the difference of two physical clocks. Finally, to be correct, we distinguish the two cases where $(pc_k(t') - pc_k(t)) \geq t^z$ and $(pc_k(t') - pc_k(t)) < t^z$. This result is later used in the justification of our approach.

In the following section, we discuss how the phenomenon of clock drift is addressed when modeling real-time systems with MECHATRONIC UML.

4 Modeling

In this section we describe MECHATRONIC UML, our modeling approach for the design and analysis of mechatronic systems. First we introduce the basic concepts of MECHATRONIC UML. Then, we consider the platform independent model (PIM) (Section 4.1) and platform specific model (PSM) (Section 4.2) and assign our modeling approaches to both levels. Afterwards we describe which information is necessary to consider systems with a bounded precision (Section 4.3).

4.1 Platform Independent Model of MECHATRONIC UML

Aspects of a platform independent model are valid for a class of platforms. E. g., deadlines are platform independent, as these are requirements of the system and not of the specific platform. In the following, we describe the concepts of MECHATRONIC UML by considering the PIM.

For the structural view, MECHATRONIC UML supports components, which are based on UML 2.0 components, and patterns, which are adapted to the domain of hard real-time component-based systems. We differentiate between discrete, continuous, and hybrid components. In general, if we say component, we mean discrete component. Figure 2 shows the structure of the air gap example (cf. Section 2). The hybrid Monitor component embeds and controls the continuous Storage component, continuous Sensor component and hybrid AirGap component. Besides the aforementioned coordination aspects of the Monitor, the communication with the Registry is realized by the Monitor, too.

Every shuttle needs to log in to a track section via the Registry. Furthermore, the Registry is used to store the information generated by a shuttle. This information can be used by following shuttles to optimize the energy consumption by optimization of the air gap. Therefore, the Monitor component sends information about a track section to the Registry and tries to get information of prior shuttles. To get the information about the track, the Sensor component is used. To store the information of prior shuttles, the

Storage component is used. The information of both, the Sensor component and Storage component, is transmitted to the AirGap component, which uses this information for the optimization of the air gap.

For the interaction between the Monitor and Registry component, the MonitorRegistry pattern is used. The introduced patterns of MECHATRONIC UML consist, in the structural view, of roles and a connector between two roles. The MonitorRegistry pattern consists of MonitorRole and RegistryRole, which are applied by the Monitor component and Registry component.

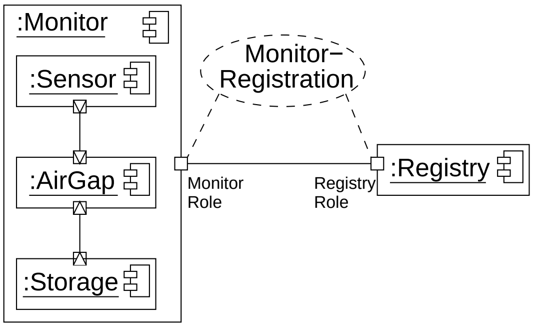


Fig. 2. Monitor and registration component and pattern

Besides the introduced structural view of MECHATRONIC UML, we now introduce the behavioral view of MECHATRONIC UML. Therefore, we first look at the patterns of MECHATRONIC UML. The behavior of a pattern role is specified by real-time statecharts. Real-time statecharts are syntactically based on statecharts. To enable the modeling of real-time systems, the syntax extensions of real-time statecharts consist of timing constraints in form of deadlines, WCET, time invariants and time guards. The semantics of real-time statecharts are based on the semantics of Timed Automata [12]. The lower and-state in Figure 3 shows the MonitorRole behavior. If the registry has no information, the MonitorRole sends requests to the RegistryRole every 20 ms. If MonitorRole gets information, these are added to the storage. Furthermore MonitorRole sends experience to the Registry.

The behavior of components is specified by hybrid statecharts. These are real-time statecharts with the possibility of embedding hybrid components where ports can be continuous, discrete or hybrid. Figure 3 shows the behavior of the hybrid Monitor component. The lower and-state is the applied MonitorRole behavior. The upper and-state specifies the so called synchronous and-state. There, all applied pattern roles are synchronized with each other to enable a valid overall behavior. Within the four states of the upper and-state, AllAvailable, AbsAvailable, NoneAvailable, and RefAvailable, continuous components are embedded. Thus, a switch to another state yields a reconfiguration, as another configuration of controllers is used. To consider the fading, we introduce a fading function. The fading function abstracts from the needed continuous fading states and specifies just the time needed for fading, as this is the important information for switching to another state.

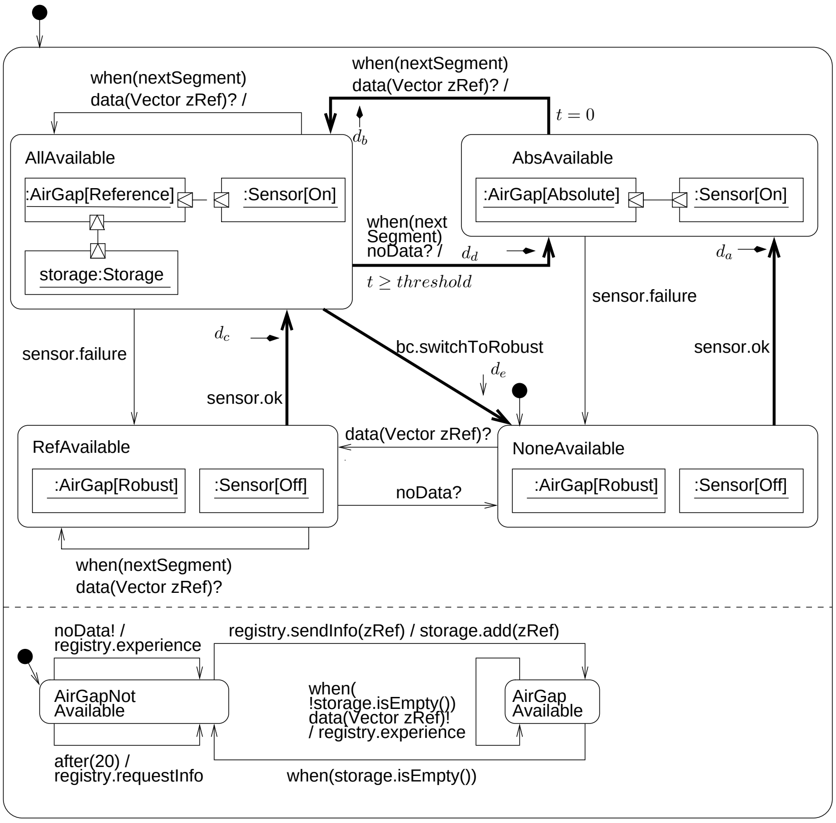


Fig. 3. Monitor behavior with embedding

To complete our example, we need a periodic interaction behavior for passing the air gap informations from one rotor to another. This can be local or distributed, as shown in Section 2. Safe periodic (time triggered) information passing is important, as an omission could lead to a failure. E. g., if a shuttle is in the RefAvailable state and omits an air gap information or omits the timeliness of the air gap information passing, the minimized air gap could lead to an unsafe behavior, as the shuttle did not recognize in time that it should switch to a safe NoneAvailable state. Figure 4 and Figure 5 specify a periodic, safe passing of the air gap information.

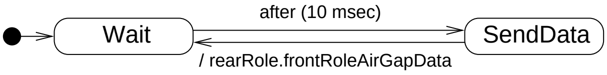


Fig. 4. Periodic information passing

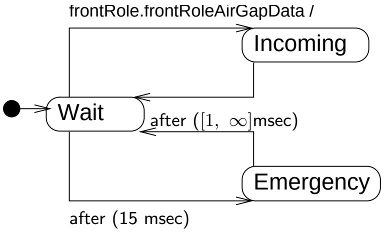


Fig. 5. Watchdog for getting periodic information

4.2 Platform Specific Model of MECHATRONIC UML

Platform specific aspects are only valid for one specific platform. E. g., the worst case execution time of a system is computed by considering the architecture of the platform and therefore is often not valid for another platform. In contrast to Section 4.1, where WCETs are specified in form of requirements, in this section WCETs are specified in form of the concrete WCETs of the generated code for a specific platform.

To determine platform specific information, we first deploy the PIM components to the given hardware infrastructure. Concrete component instances are assigned to dedicated nodes and cross node links are assigned to networks. Afterwards we use this deployment to get platform specific information, which are specified in a so called platform model (PM) [13]. In a platform model, relevant characteristics such as CPU type and operating system are described. Knowing all relevant elements of a platform, we can employ WCET techniques as described in [14]. These values can be taken for elementary instructions. Furthermore, the determined WCET of elementary elements can than be taken to determine the WCET of complex methods of the real-time statecharts by summing up this elementary instructions.

To get the final platform specific model, we have to take into account the aforementioned information and map the components and links to threads and network links.

4.3 Modeling with Clock Drift

In the introduced time dependent interaction behavior (cf. Section 4.1), a precision (cf. Definition 6) or clock drift is not considered. This could lead to malfunction. Consider the following example: If the offset between the shuttles or components is 6 ms, then the specified communication behavior in Figure 5 switches to state Emergency.

As mentioned in Section 3, a realistic clock rate is 10^{-5}. This clock rate implies a clock drift of $9999/10000$ to $10001/10000$. Therefore, for an arbitrary large interval, two clocks drift infinity also if the clock drift is bounded. Considering clock drifts for modeling, we need a bounded precision in addition to the clock rate. As only the bounded precision and not clock drifts are take into account for the system analysis, we cannot simply apply the UML Profile for Schedulability, Performance, and Time [15] here as it only consider the drift. Therefore, we capture instead the information about the precision of the system Π during modeling, which enables the verification of the system considering this precision as outlined later in Section 5.

To enable the a priori detection of failures, which are based on different clock drifts yielding large offsets, we add modeling constructs for the offsets to our behavior models. We differ between the PIM and PSM level.

On the PIM level, a maximal assumed precision Π can be added to our pattern behavior (cf. Figure 6). Then, the pattern, with the maximal assumed precision, can be verified by model checking (cf. Section 5).

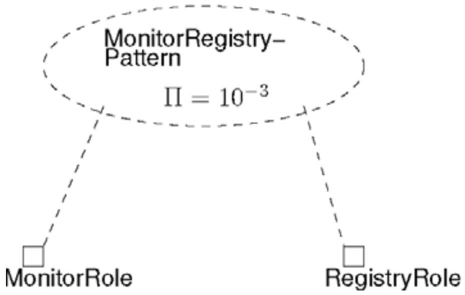


Fig. 6. Adding precision to MECHATRONIC UML

At the PSM level, the concrete clock drifts are derived for the applied pattern considering the concrete platform. Therefore, the maximal offsets are specified in the deployment model (cf. Figure 7).

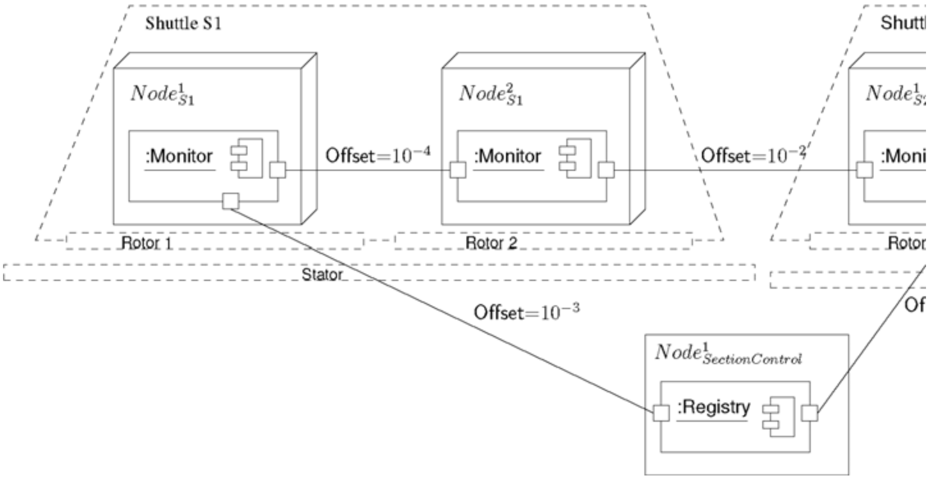


Fig. 7. Deployment with offset

If the derived offset for a pattern instance is smaller than the precision specified at PIM level, the platform independent verification of the behavior at the PIM level also holds for the PSM level. If, for example, $\mathit{offset}_t^{Node^1_{S1},Node^2_{S1}} \leq \Pi$, all properties verified for the pattern also hold for the concrete pattern instance, as a better precision does not invalidate them. Otherwise, if the precision in the PSM is lower than assumed in the PIM, we need to verify the pattern with a precision $\Pi \geq \mathit{offset}$. For example,

$\mathit{offset}_t^{Node^2_{S1},Node^1_{S2}} \geq \Pi$ and therefore the pattern needs to be verified with a $\Pi \geq \mathit{offset}_t^{Node^2_{S1},Node^1_{S2}}$.

5 Verification

In the last section, we outlined our design approach for real-time systems using MECHATRONIC UML. As mentioned, compositional verification is one distinguishing capability in the approach. The following subsection deals with the existing verification approach [11, 10], and the limitations concerning the problem of clock drifts are discussed (cf. Section 5.1). Taking into account these limitations, we extend our existing verification techniques such that the known upper bound clock precision is taken into account (cf. Section 5.2).

5.1 Verification without Clock Drift

Our modeling approach supports components and patterns as structural modeling concepts. All behavioral aspects are modeled with real-time statecharts or their hybrid extension hybrid reconfiguration charts [9, 11].

Due to the compositional nature of our approach (cf. [9]), each pattern and component can be verified separately. Thus, after designing each pattern, these can be verified for themselves. At the same time, for each component can be verified whether it fulfills all roles needed. For the composition of the system, it is not necessary to conduct another verification step as the correct behavior can be inferred from the other verification steps.

For the verification which does not take into account the effects of clock drifts, the statecharts models and their time related extensions are mapped to the formal model of Timed Automata [8, 10] and the real-time model checker UPPAAL is called upon for the verification task. In this mapping, all timing annotations from the real-time statecharts are mapped without any loss of information to the Timed Automata model. For this reason, we will later argue at the formal model of Timed Automata directly rather than taking the syntactical more complex real-time statechart syntax into account.

When checking a single pattern, the parallel composition of all Timed Automata for the role behavior as well as the connectors results in a closed model which is verified.

The situation is slightly more complex for components. The orthogonal states in the real-time statecharts of the component result in parallel Timed Automata which include the refined role behaviors as well as a synchronization part which contains unconnected external events and is thus open (environment operates non-deterministically).

To ensure that the component refines each of the role protocols associated to its ports, we propose either to use syntactical refinement rules which ensure this when checking the component model for the absence of deadlocks or alternatively model checking can be employed to also fully automate this task (cf. [11, 16]).

This straight forward application of time automata verification technology of our model without taking into account clock drifts does only answer the questions whether the model executed on a perfect hardware without clock drift fulfill the checked conditions. While this is a reasonable first step and very often their are good arguments the effects of clock drift can be neglected for a specific system. However, in many cases it is not clear whether the behavior cannot have adverse effects not present in the idealized model due to clock drift.

Following the modeling of clock drift effects for the platform model and the PIM/PSM models as outlined in Section 4, a really accurate verification on the different model levels requires that we are taking the known upper bound for the clock precision into account.

5.2 Verification with Clock Precision

To eliminate the limitations of the verification without considering clock drifts we discuss verification including clock drifts in this section. Due to our domain of mechatronic systems we have reactive behavior which is modeled using real-time statecharts. Each real-time statechart has a predefined start state. From this it follows that we have to distinguish the following two cases. First the initial phase and second the clock drift during runtime.

Controlling initialization of clocks

The first obvious reason for clock drifts in distributed systems is that clocks usually do not start at the same point of time on each node. This is because the clocks on each node do not have any reference to a global clock and hence each other. Discussing this on the model level, each parallel state chart decides non-deterministically when to start. A priori this effects an unbounded clock drift. Due to the use of coordination patterns in our MECHATRONIC UML approach (cf. Section 4), we are able to control the critical initial phase.

In [17] we introduce an technique integrated in the MECHATRONIC UML approach for this concept. Here it is possible to control the point of initialization of the pattern and specify rules which the corresponding pattern has to fulfill. E. g. if an upper bound for the overall inital phase is specified hence we know die initial difference between all involved clocks. Furthermore if the counterpart does not preserve this upper bound, fail-safe behavior is guaranteed by the protocol of the coordination pattern.

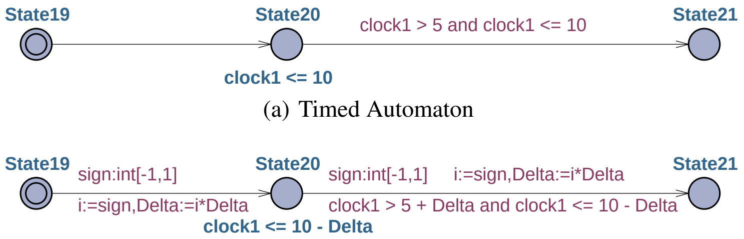
Clock precision during runtime

As mentioned before, clocks can drift away from each other during runtime. To take this into account on the model level, e. g. when verification is performed, we have to emulate the behavior of drifts in the corresponding model. Therefore we first have to identify all timing elements in the model. In our formal model of Timed Automata, *timeguards* and *invariants* as all the relevant elements (cf. Definition 7).

To emulate clock drift, we propose to map the original model to a Timed Automaton model where we take the precision into account by realizing the best and worst-case offset $\pm Delta$ of the timeguards and invariants in a non-deterministic manner. Such an encoding results in a pessimistic abstraction from the concrete offsets between clocks.

In general, for the mapping of a timeguard tg_i of a transition $trans_i$, the precision $+Delta$ or $-Delta$ is added (in more detail: $clock \leq const_{lower} - Delta$ or $clock \geq$

$const_{upper} + Delta$). The same applies for invariants. To model the required non-deterministic choice in a compact manner, we use a variable $sign$ with domain $\{-1, 1\}$ as a random coin. In each transition entering a location with a clock invariant or an outgoing transition with time guard, we non-deterministically chose whether the setting of the offset $Delta$ is positive or negative ($Delta := -Delta$ or $Delta := +Delta$).



(a) Timed Automaton

(b) Corresponding Timed Automaton including clock drift

Fig. 8. Example for taking clock drift into account

In the following we will explain the integration of clock drift in the model on the basis of an example. In Figure 8(a) a simple Timed Automaton is shown. The Timed Automaton consists of three locations (*State19*, *State20*, and *State21*) and a local clock *clock1*. *State20* ist associated with a clock invariant $clock \leq 10$ and the transition *State20* → *State21* is associated with a timeguard $clock1 > 5$ and $clock1 \leq 10$. Since the local clock diverges, we have to model the offset *Delta* by extending the model when e. g. verification is performed. The corresponding model which is used for e. g. verification is depicted in Figure 8(b). As explained above we have to extend all invariants and timeguards in this model.

Due to the pessimistic nature of the proposed emulation, we can conclude that the behavior with clock drift behaves correct if we have proven this for the proposed abstraction. However, the pessimistic abstraction can also result in problems encountered in the abstraction which cannot happen if the exact clock behavior would have been taken into account (thus results are often named false negatives).

One particular case of such false negatives can be already detected when extending the guards: If the offset is too large in comparison to the timeguard bounds, the pessimistic abstraction in fact results in *empty timeguards* which cannot be fulfilled. For the timeguards $clock <= const - Delta$ and $clock => const' + Delta$ this is the case when $const - const' \ngeq 2Delta$.

We further assume that the clock precision is at least twice as good as the expected accuracy for the activation (which equals $const - const'$) and further not discussed in this paper those cases where this condition is not fulfilled (e.g., when exact timing has been specified: $const = const'$).

5.3 Evaluation

An evaluation of our approach for the verification of models containing clock drifts is given in this subsection. As an evaluation example we use the *air gap transmission* example displayed in Figure 4 and Figure 5. We check the ATCTL properties A[] not deadlock (1) and A<> Emergency (2).

First, we verified both properties for our model without taking drifts into account explicitly. With each the verification for property (1) and (2) took about 1 second and at maximum about 4000 KB were allocated by the verifier[4].

Afterwards we performed the mapping from the model without clock drift to the model with clock drift. As expected, both properties are fulfilled again. Due to the enriched model with variables, the verification took about 1 second again and a maximum of about 7050 KB were allocated by the verifier.

6 Justification

In this section, we justify that the emulation scheme informally outlined in Section 5.2 is indeed sufficient to verify the correctness of a Timed Automaton model where clock drift and clock precision is taken into account. In addition, we proof that a check for an emulation with lower precision always imply that the same ATCTL properties also hold for the model with higher precision, which is exploited when we reuse verification steps for pattern witch have been done with the *required* precision to conclude that concrete pattern instances are correct for a given precision derived from the platform model which is higher than the required one.

For this purpose, we first introduce a simpler Timed Automaton model in Section 6.1 and define related semantic without clock drift (Section 6.2) and with clock drift (Section 6.3). Then, we provide the proofs for the step from the one without clock drift to the one with clock drift (Section 6.4) as well as between emulation version with higher and lower precision (Section 6.5).

6.1 A Simplified Timed Automata Model

For simplification and for a better readability of the proof, we introduce a simplified model of Timed Automata:

Definition 7. *A* timed automaton *is described by a 7-tuple* $(L, V^c, I, O, T, C, S^0)$ *with* L *a finite set of locations,* V^c *a set of clocks,* I *a finite set of input signals,* O *a finite set of output signals,* T *a finite set of transitions,* $C : L \to [[V^c \to \mathbb{R}] \to \{true, false\}]$ *a clock constraint, and a set of initial states* $S^0 \subseteq \{(l, x) | l \in L \wedge x \in [V^c \to \mathbb{R}] : (C(l))(x)\}$. *For any transition* $(l, g, g', a, l') \in T$ *holds that* $l \in L$ *is the source-location,* $g \in COND(V^c)$ *the time guard,* $g' \in \wp(I \cup O)$ *the I/O-guard,* $a \in [[V^c \to \mathbb{R}] \to [V^c \to \mathbb{R}]]$ *the clock update, and* $l' \in L$ *the target-location.*

Based on this simplification, we will define a related semantics without clock drift (Section 6.2) and with clock drift (Section 6.3).

[4] The verification was done on a Pentium 4, 2.4 GHz, 1 GB memory, OS Linux Redhat.

6.2 Idealized Standard Semantics

To model the state of an abstract Timed Automaton, we use an element x of the set of possible clock variable bindings $X_c = [V^c \rightharpoonup \mathbb{R}]$ and a discrete state l (named location) of the set of all possible location L. The state of a system can then be described by a pair $(l,\, x) \in L \times X_c$.

For $X_c = [V^c \rightharpoonup \mathbb{R}]$ the set of possible clock variable bindings, the inner state of a Timed Automaton can be described by a pair $(l,\, x) \in L \times X_c$ with $x \in [V^c \rightarrow \mathbb{R}]$. There are two possible ways of state modifications: Either by firing an instantaneous transition $t \in T$ changing the location as well as the state variables or by residing in the current location which consumes time and just increases the clock variables.

When staying in state $(l,\, x)$ firing an instantaneous transition $t = (l',\, g,\, g^i,\, a,\, l'')$ is done iff $l = l'$ (the transitions source location equals the current location) and the clock guard is fulfilled ($g(x) = true$), the I/O-guard is true for the chosen input and output signal sets $i \subseteq I$ and $o \subseteq O$ ($i \cup o = g^i$), and $a(x) \in C(l'')$. The resulting state will be $(l'',\, a(x))$ and we note this firing by $(l,\, x) \rightarrow_{(i \cup o)} (l'',\, a(x))$.

If no instantaneous transition can fire, the Timed Automaton resides in the current location l for a non-negative and non-zero time delay $\delta > 0$. In state $(l,\, x)$ such a time consuming transition may result in $(l,\, x \oplus \delta)$ if all $\delta' \in [0, \delta]$ holds $C(x \oplus \delta)$.[5]

The trace of a timed behavior is a possibly infinite execution sequences $(l_0, x_0, \delta_0) \rightarrow_{a_0} (l_1, x_1, \delta_1) \ldots$ where all $(l_i, x_i \oplus \delta_i) \rightarrow_{a_i} (l_{i+1}, x_{i+1})$ are valid instantaneous transition of the system state and $(l_i, x_i \oplus \delta'_i)$ are valid for all $\delta'_i \in [0, \delta_i]$.

To abstract from the concrete abstract Timed Automaton state, we can consider only the externally visible timed path $\pi = (\delta_0); a_0; \ldots; (\delta_n); a_n$ such that we write $(l_0, x_0(0)) \rightarrow_\pi (l_n, x_n(\delta_n))$ iff $(l_0, x_0, \delta_0) \rightarrow_{a_0} (l_1, x_1, \delta_1) \ldots (l_n, x_n, \delta_n)$. The offered interactions for a state (l, x) are further denoted by the set offer$(M, (l, x))$ which is defined as $\{a | \exists (l', x') : (l, x) \rightarrow_a (l', x')\}$.

Each abstract Timed Automaton M_i thus is related to a unique timed transition system $\mathcal{M}_i$ by the state space $L_i \times X_i$ and timed path between these states denoted by $[\![M]\!]$. An appropriate notion of real-time refinement can then be defined for timed transition systems as follows:

Definition 8. *For two timed transition systems $\mathcal{M}_I$ and $\mathcal{M}_R$ holds that $\mathcal{M}_R$ is a* refinement *of $\mathcal{M}_I$ denoted by $\mathcal{M}_R \sqsubseteq_{RT} \mathcal{M}_I$ iff for every initial state c of $\mathcal{M}_R$ an initial state c'' of $\mathcal{M}_I$ exists with*

$$\forall c \Rightarrow_\pi c' \quad \exists c'' \Rightarrow_\pi c''' \tag{2}$$

$$\forall c \Rightarrow_\pi c' \quad \exists c'' \Rightarrow_\pi c''' : \quad \mathit{offer}(\mathcal{M}_R, c') \supseteq \mathit{offer}(\mathcal{M}_I, c'''). \tag{3}$$

As refinement is a precongruence for $\|$, we can exclude time-stopping deadlocks etc. by looking only into more abstract behavior ($\mathcal{M}_I$) instead of their refinements. In addition, refinement ensures that all traces possible in the refining behavior ($\mathcal{M}_R$) are also possible in the the refined one ($\mathcal{M}_I$).

[5] The clock assignment $x \in X$ continuously increases and we denote an increment δ by the assignment $x \oplus \delta$.

6.3 Semantics with Clock Precision

In accordance with Definition 1, we can assume that each Timed Automaton M_k has a physical clock pc_k rather than the reference clock z as basis. To model the state of an Timed Automaton when the clock drift and offset implied by the physical is taken into account, we can use clock variable bindings X_c and the discrete states (locations) L like in the idealized case. However, the traces have to be different, as we do not make the assumption that the clock values evolve perfectly synchronized with z. Therefore, we have to employ explicit trajectories $\rho : [0, \delta] \to [V^x \to \mathbb{R}]$ for a time period δ and only have to restrict these trajectories such that for all $t \geq 0$ with $t \leq \delta$ the precision Π is respected.

A trace for a timed behavior M with a a given precision Π (cf. Definition 6 and Equation 1) is thus a possibly infinite execution sequences $(l_0, \rho_0, \delta_0) \to_{a_0} (l_1, \rho_1, \delta_1) \to_{a_1} (u_2, l_2, \rho^2_{u_2}, \theta^2_{u_2}, \delta_2) \ldots$ of M.

To abstract from the concrete Timed Automaton state, we can again use the concept of a timed path $\pi = (\delta_0); a_0; \ldots; (\delta_n); a_n$ such that we write $(l_0, \rho_0(0)) \to_\pi (l_n, \rho_n(\delta_n))$ iff $(l_0, \rho_0, \delta_0) \to_{a_0} (l_1, \rho_1, \delta_1) \ldots (l_n, \rho_n, \delta_n)$.

Each abstract Timed Automaton M_i with precision Π is thus is also related to a unique timed transition system with the state space $L_i \times X_i$ and all timed path between these states denoted by $[\![M]\!]^{\Pi}_{cd}$ by taking also the possible clock drift into account. Therefore we can also employ the refinement notion of Definition 8 to compare timed behaviors. Most importantly for our considerations, we are also able to compare timed behavior without clock drift with timed behavior with clock drift by referring to the timed path only.

6.4 Emulation Theorem

In general, for an abstract Timed Automaton M holds that the behavior $[\![M]\!]$ can be quite different from $[\![M]\!]^{\Pi}_{cd}$. The clock drift rate can result in additional traces as well as time-stopping deadlocks.

For the Timed Automaton construction which emulates the effects of drifts and offsets introduced in Section 5, we have claimed that the constructed automata can be considered to exclude errors in its realization on a system with known upper bounds for the clock drift rates. We will prove in the following that the emulating behavior without clock drift and offset is refined by the original behavior with bounded clock drift and offset as defined in the last subsection and thus can be studied as a valid abstraction to identify erroneous behavior.

Theorem 1. *For an abstract Timed Automaton M and its emulation version $E(M, \Pi)$ for precision Π holds that $[\![M]\!]^{\Pi}_{cd}$ is a* refinement *of $[\![E(M, \Pi)]\!]$*

$$[\![M]\!]^{\Pi}_{cd} \sqsubseteq_{RT} [\![E(M, \Pi)]\!].$$

Proof. We can prove that condition 2 is fulfilled for a state (l, x) in M and (l_E, x_E) of $E(M, \Pi)$, by induction over the length of any path.

For length zero, trivially the condition is fulfilled. Furthermore, the precision Π is true for M and $E(M, \Pi)$.

When extending a trace of length n by another step, we have to consider either a instantaneous or a timed step.

In the first case for any transition via event e_0 the construction of the emulation ensures that a related state in $[\![E(M,\Pi)]\!]$ can be reached via a positive sign step which offers the related transition (same clock update and leads to the related state) as for the time guards $v \leq c$ a time guard $v' \leq c \pm \Pi$ exists which must be enabled if $v \leq c$ is enabled. As the transition and its related transitions have the same clock resets, the precision Π is also true for the reached state.

In the second case, the construction of the emulation ensures that a state in $[\![E(M,\Pi)]\!]$ exists which invariant permits to stay δ_0 time within the related state (same transitions) as for the time invariant $v \leq c$ a time invariant $v' \leq c \pm \Pi$ exists which must be true if $v \leq c$ is true. After a time step δ_0, it follows directly from applying Equation 1 that the precision Π must be true for the reached state, too.

To also show that condition 3 always holds for a state and its emulated counterpart, we can use the fact that a time guard $v \leq c$ is only blocked if also a related state is reached via the negative sign case where the time guard $v' \leq c \pm \Pi$ is also blocked.

Therefore, we have shown that our approach to consider clock drift by emulating it ensures that the idealized semantics of the emulation model is refined by the non idealized semantics with clock drift and offset of the original model. This refinement relation ensures that all possible traces of $[\![M]\!]^{\Pi}_{cd}$ are also possible in $[\![E(M)]\!]$ and that all deadlocks possible in $[\![M]\!]^{\Pi}_{cd}$ are also possible in $[\![E(M)]\!]$. Therefore, we can check $[\![E(M)]\!]$ with a standard model checker for Timed Automata as described in Section 5 and conclude that the verified ATCTL properties as well as deadlock freedom also hold for $[\![M]\!]^{\Pi}_{cd}$ which could not be checked directly.

6.5 Precision Theorem

As outlined in the Section 4 for modeling, it is only necessary to verify concrete pattern instances if the concrete precision is worse than the one assumed at the PIM level. To justify this obvious rule, we will prove in the following that an emulating behavior without lower precision Π is always refined by an emulating behavior with higher precision Π' ($\Pi \geq \Pi'$).

Theorem 2. *For an abstract timed automaton M and its emulation versions $E(M,\Pi)$ with precision Π and its emulation version $E(M,\Pi')$ with precision Π' holds for $\Pi \geq \Pi'$ that $[\![M_E E(M,\Pi')]\!]$ is a* refinement *of $[\![E(M,\Pi)]\!]$:*

$$\Pi \geq \Pi' \Rightarrow [\![E(M,\Pi')]\!] \sqsubseteq_{RT} [\![E(M,\Pi)]\!].$$

Proof. We can analogously to Theorem 1 prove that condition 2 is fulfilled by induction over the length of any path showing that the lower precision will include all traces of the emulation with higher precision.

To also prove condition 3, we can use the fact that a time guards is only blocked in the version with less precision, if it may also be blocked in the model with higher precision.

This second theorem permits to safely skip verification tasks if they have been successfully done for a model with lower precision than the concrete precision present in the model by exploiting the fact that the emulation scheme for a higher precision is a refinement of one with a lower one.

7 Related Work

To the best of our knowledge no work on the systematic and automated consideration of clock drift during the verification of Timed Automata exists.

There are cases such as [7] where the verification of a specific problem took clock drift into account exploiting the specific characteristics of the problem into account. In the presented work in contrast, the model remains unchanged and the required extension of the timed model reelecting the effects of clock drift are woven into it automatically.

In [6] Timed Automata are employed to prove that a specific periodic clock synchronization protocol for a CAN bus guarantees a given precision under certain fault assumptions. In contrast, the presented approach verifies whether application specific correctness criteria hold given an upper bound for the precision without making any assumptions about the specific clock synchronization approach. Even in the case that the specific clock synchronization protocol is known, it seems that to encode the application specific time constraints on top of the explicit modeled time, Timed Automata with additive clock constraints are required. However, Timed Automata with addition of clock values in the time constraints cannot be checked as their emptiness problem is undecidable (cf. [8]).

There exist a few approaches which discuss the problem of clock drift on the level of model based development. We first discuss some approaches which take clock drift into account on the model level. In the second part we discuss some tools which abstract from clock drift for some reason.

In the *UML Profile for Schedulability, Performance, and Time* [15], the integration of clock drift is introduced. This profile allows the explicit modeling of *drift* and *skew*.

In the following, the to ours most similar approach, is discussed. In [18], an approach for real-time modeling in UML, focusing on analysis and verification of time and scheduling related properties is introduced. To reach this aim, a concrete UML profile, called the *OMEGA-RT profile*, is defined, dedicated to real-time modeling by identifying a set of relevant concepts for real-time modeling, which can be considered as a refinement of the standard [15] profile. In [18], the existence of a global reference time is supposed. As proposed in the standard profile, local time can be defined by means of local clocks, for which a maximal drift and/or offset with respect to global time can be defined [18].

Both approaches, the *UML Profile for Schedulability, Performance, and Time* and the *MEGA-RT profile*, do not take into account the problem of clock drift in the verification task and thus the PSM level is not addressed adequately.

The *de facto* industry standard for modeling of mechatronic systems with hybrid behavior is MATLAB/Simulink and Stateflow[6]. Formal verification of MATLAB/Simulink and Stateflow models can be performed by using the model checker

[6] http://www.mathworks.com

CheckMate. Besides MATLAB/Simulink, there exists a number of tools which also integrate model checking. CHARON [19] and UPPAAL [20] provide, on the one hand, support for the modeling of real-time systems. On the other hand, the verification using model checking does not take into account clock drifts in all mentioned tools.

Finally, Henzinger et al. introduce in [21] the formalism of *robust Timed Automata*, which are Timed Automata that accept all trajectories *robustly*. Basically, in this approach other problems are mainly discussed. However, they stated that it is possible to set the used ϵ on a minimum as required and therefore the concept of robust Timed Automata can be used in this context.

8 Conclusion and Future Work

In our paper we present an approach for the verification of real-time systems using the MECHATRONIC UML, taking the problem of clock drift and clock offsets into account. The paper at first presents an extension of the MECHATRONIC UML approach which supports capturing the required platform characteristics in the different UML model elements. Secondly, a scheme for the emulation of the clock drift effects has been presented and justified, showing that the emulation results in a valid abstraction and thus permits to prove required safety properties.

The considered example and the first verification results indicate that the additional effort required for a verification which takes clock drift into account is often feasible. Also the case that the pessimistic abstraction results false negatives such as time-stopping deadlocks due to the modifications of the clock constraints which applies only for very small time intervals which are smaller than the precision seem to occur only in rare cases. However, we plan as future work to more systematically study for which model characteristics the proposed emulation scheme can be effectively applied and which alternative encoding schemes might be applicable when the rewriting of the time constraints cannot be done without introducing obvious false-negatives. In addition, a more complete analysis of the complexity of the underlying verification problem is planned.

References

1. Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.
2. Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML models via a mapping to communicating extended timed automata. In *Model Checking Software: Proceedings of the 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, Barcelona, Spain, 2004. Springer Verlag.
3. Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.
4. Hermann Kopetz, Astrit Ademaj, and Alexander Hanzlik. Combination of clock-state and clock-rate correction in fault-tolerant distributed systems. *Real-Time Syst.*, 33(1-3):139–173, 2006.

5. Alexander Hanzlik. A case study of clock synchronization in flexray. Research Report 31/2006, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2006.
6. Guillermo Rodriguez-Navas, Julian Proenza, and Hans Hansson. An uppaal model for formal verification of master/slave clock. In *6th IEEE Int'l Workshop on Factory Communication Systems (WFCS)*, Torino, Italy, June 2006. IEEE Electronics Society.
7. Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.
8. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
9. Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.
10. Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In *Proceedings of the International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, October 2004.
11. Holger Giese and Martin Hirsch. Modular verificaton of safe online-reconfiguration for proactive components in mechatronic uml. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops, Montego Bay, Jamaica, October 2-7, 2005*, volume 3844 of *Lecture Notes in Computer Science (LNCS)*, pages 67–78. Springer Verlag, 2006.
12. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
13. Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, Lecture Notes in Computer Science (LNCS), pages 25–40. Springer Verlag, November 2005.
14. Edwin Erpenbach. *Compilation, Worst-Case Execution Times and Scheduability Analysis of Statechart Models*. Ph.D.-thesis, University of Paderborn, Department of Mathematics and Computer Science, February 2000.
15. Object Management Group. *UML Profile for Schedulability, Performance, and Time, v1.1*, 2005.
16. Holger Giese and Martin Hirsch. Checking and Automatic Abstraction for Timed and Hybrid Refinement in Mechtronic UML. Technical Report tr-ri-03-266, University of Paderborn, Paderborn, Germany, December 2005.
17. Holger Giese, Sven Burmester, Florian Klein, Daniela Schilling, and Matthias Tichy. Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In B Henderson-Sellers and J Debenham, editors, *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies*, pages 21–32, Anaheim, CA, USA, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia, October 2003.
18. Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for UML. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(2):113–127, August 2004.

19. Franjo Ivancic. *Modeling and Analysis of Hybrid Systems*. PhD thesis, University of Pennsylvania, 2003.
20. K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology*, 1(1), 1997.
21. Vineet Gupta, Thomas A. Henzinger, and Radha Jagadeesan. Robust timed automata. In *HART '97: Proceedings of the International Workshop on Hybrid and Real-Time Systems*, pages 331–345, London, UK, 1997. Springer-Verlag.

Analyzing Robustness of UML State Machines

Steffen Prochnow, Gunnar Schaefer, Ken Bell, and Reinhard von Hanxleden

{spr,gsc,kbe,rvh}@informatik.uni-kiel.de
Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, D-24118 Kiel, Germany

Abstract. State Machines constitute an integral part of software behavior specification within the UML. The development of realistic software applications often results in complex and distributed models. Potential errors can be very subtle and hard to locate for the developer. Thus, we present a set of *robustness rules* that seek to avoid common types of errors by ruling out certain modeling constructs. Furthermore, adherence to these rules can improve model readability and maintainability. The robustness rules constitute a general Statechart style guide for different dialects, such as UML State Machines, *Statemate*, and *Esterel Studio*. Based on this style guide, an automated checking framework has been implemented as a plug-in for the prototypical Statechart modeling tool *KIEL*. Simple structural checks can be formulated in a compact, abstract manner in the OCL. The framework can also incorporate checks that go beyond the expressiveness of OCL by implementing them directly in Java, which can also serve as a gateway to formal verification tools; we have exploited this to incorporate a theorem prover for more advanced checks. As a case study, we have adopted the UML *well-formedness rules*; this confirms that individual rules are easily incorporated into the framework.

1 Introduction

Statecharts [1] constitute a widely accepted formalism for the specification of reactive real-time systems. They extend the classical formalism of finite-state machines and state transition diagrams by incorporating notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior. They have been incorporated into the Unified Modeling Language (UML) (as object-based State Machines) [2] and are supported by several commercial tools, *e. g.*, *Esterel Studio*[1], *Matlab/Simulink/Stateflow*[2], and UML *CASE* tools, such as *Rhapsody*[3] and *ArgoUML*[4].

[1] http://www.esterel-technologies.com/products/esterel-studio/
[2] http://www.mathworks.com/products/stateflow/
[3] http://www.ilogix.com/
[4] http://argouml.tigris.org/

The assurance of quality, *i. e.*, ensuring readability and avoiding error-prone constructs, is one of the most essential aspects in the development of safety-critical reactive systems, since the failure of such systems—often attributable to programming flaws—can cause loss of property or even human life. As Parnas has observed, human code reviews are time-consuming and highly undependable in revealing errors [3]. Taking part of the burden off the reviewers, as well as off the designers, is the rationale for *automated error prevention*, where a computer performs preliminary checks. Hence, this paper is a contribution to ensure certain aspects of safety in developing Statecharts. We achieve this by applying methods of automated error-source detection.

We propose a rule set that forms a fundamental Statechart style guide. Based on this well-structured set of robustness rules, both syntactic and semantic, an automated checking framework has been implemented as a plug-in for the *Kiel Integrated Environment for Layout* Statechart modeling tool[5]. A key objective in devising the rule set and in designing the checking framework was to not restrict the modelers' creativity, but to achieve more explicit, easy to comprehend, and less error-prone models. Our approach, therefore, was developed adhering to the following requirements:

Modularity and Configurability: All robustness checks are independently implemented, individually selectable, and parametrizable via a preferences management.

Extendability of the rule set: The set of checks is easily extendable by either adding a constraint, specified in the Object Constraint Language (OCL) [4], or by implementing a new Java class.

Automatic conformance checking: Compliance with the robustness rules can be checked very rapidly—a key quality, imperative for end-user acceptance. Due to the uncoupling of the checking process from the modeling process, the checks may be applied at all stages of system development, even to partial system models.

Even though a wide range of applications for Statechart verification already exists, none fulfills all needs. They are either highly specialized and therefore not extendable or they are extendable but do not provide the possibility to check complex problems. In contrast, we present a general approach to style checking in Statecharts. It is easily extendable and incorporates a theorem prover to provide for complex semantic checks. The main contributions of this paper are:

- An inspection and classification of error prevention methods for software in general as well as with a focus on style checking in Statecharts (Section 3),
- a comparison of existing style guides and applications for textual programming languages and Statecharts (Section 3.3),
- a collection of *robustness rules* for less error-prone Statecharts (Section 4),
- a checking framework, which automatically and quickly evaluates rules defined in OCL; additionally, checks based on theorem proving are evaluated (Section 5), and

[5] `http://www.informatik.uni-kiel.de/~rt-kiel/`

– an experimental evaluation, based on the aforementioned checking rules, showing the applicability and efficiency of our robustness rules (Section 7).

2 Related Work

Error prevention in software development is as old as the field of software development itself. Therefore, many style guides for classical textual programming languages have been developed, dealing not only with code layout, but also with robustness aspects; *e. g.,* MISRA proposed a C programming style [5], Sun proposed a Java programming style[6]. Style guides have been developed for the Statecharts modeling paradigm as well, *e. g.,* by the MathWorks Automotive Advisory Board (MAAB)[7] and by Ford Motor Company[8], both exclusively for *Simulink*/*Stateflow*. Scaife *et al.* [6] propose the development of a *safe subset* of the *Stateflow* language, which is considered to be less error-prone. Furthermore, Kreppold [7] has presented a style guide for *Statemate*.

In the context of UML State Machines, the *well-formedness rule*s defined within the UML specification clarify the semantics of Statechart elements. Besides the *well-formedness rule*s, other rules for UML State Machines were formulated, *e. g.,* by Mutz [8]. For our style guide we pick up some of these rules; furthermore, we specify dialect independent as well as dependent rules inspired by different sources and add rules based on our own experience.

Automatically checking robustness (or soundness) of UML State Machines is an active field of research. Pap *et al.* [9] have investigated applicable approaches. The presented techniques include checks based on the OCL, graph transformation, special programs and finally reachability analysis driven tests. Richters [10] has investigated different frameworks that can be used when it comes to working with OCL.

A wide range of available CASE tools provide OCL support, which is generally limited to gathering constraints. Beyond, Mutz and Huhn [8, 11] have developed the *Rule Checker* for the automated analysis of user-defined design rules on UML State Machines. They pursue an interpreter-based analysis for the evaluation of OCL. However, an interpretative approach is generally considered less flexible and slower than an executive. Additionally, simple syntactic checks are executed by Java programs. No sophisticated checks involving a theorem prover are performed.

Another approach to check the style guide conformance of Statecharts is *Mint*[9] by Ricardo which is focused on the MAAB style guide. The checker primarily aims at achieving a consistent look-and-feel, enhancing readability, and avoiding common modeling errors. The *Guideline-Checker* [12], coded in *Matlab*, is a no-cost/academic alternative to *Mint*. The range of the *Guideline-Checker*

[6] `http://java.sun.com/docs/codeconv/`

[7] `http://www.mathworks.com/industries/auto/maab.html`

[8] `http://vehicle.berkeley.edu/mobies/papers/stylev242.pdf`

[9] `http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint`

is currently constricted to the most trivial checks, *e. g.*, “A [state] name does not include a blank,” or “A [state] name consists of [at least] 3 characters” [12, page 26].

Moreover, special programs for the detection of specific problems have been developed. Here, the *State Analyzer* [13], developed within DaimlerChrysler’s R&D, is a prototypical software tool to check the “determinism” of *Statemate* Statecharts. Performing an automated robustness analysis of requirements specifications, the tool verifies that for every state, the predicates (trigger and condition) of multiple outgoing transitions are pairwise disjoint. The approach for detecting non-determinism employs automated theorem proving (cf. Section 5), *i. e.*, proving the satisfiability of a formula consisting of the conjunction of each pair of transition predicates. Approaches analyzing requirements specifications are introduced by, *e. g.*, Heitmeyer *et al.* [14] for the Software Cost Reduction (SCR) formalism; Heimdahl and Leveson [15] present a similar approach for the Requirements State Machine Language (RSML).

In summary, none of the discussed methods and tools fulfill all of our needs. Therefore, we present a Statechart robustness analysis approach, based on the execution of Java code synthesized from OCL rules, that combines the usability and flexibility of OCL and—beyond the approach of Mutz and Huhn—the mightiness of a theorem prover.

3 Errors and Error Prevention in the Modeling of Statecharts

To support the early detection and elimination of modeling errors, a design methodology must provide effective communication among the various design stages of the product. This section gives an overview of common error sources in developing Statecharts and how these may be avoided.

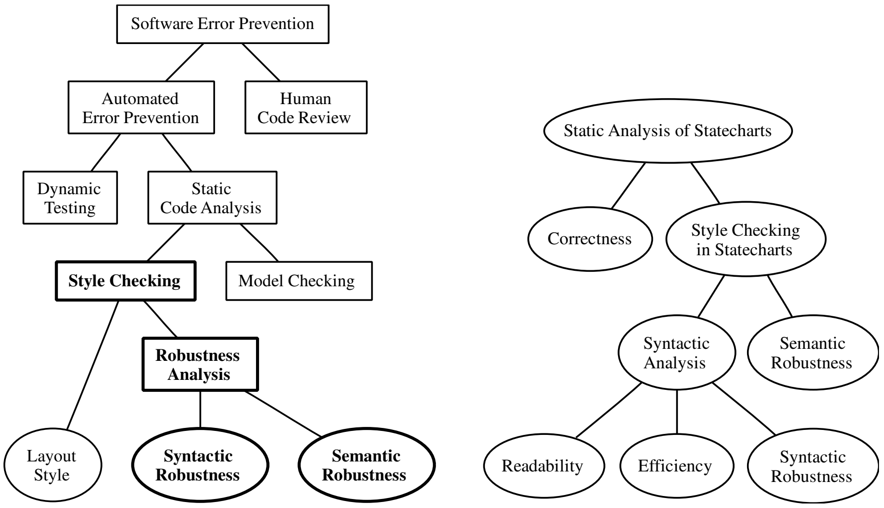
3.1 Sources of Errors

Errors in development of graphical models like Statecharts have a large diversity of types and reasons. A paramount cause of producing erroneous Statecharts is apparently a misunderstanding of the utilized modeling tools and their simulation behavior. This may have its source in *counterintuitive specifications* of the model semantics (*e. g.*, unbound behavior) and a lacking comprehension of the modeler.

Errors also originate from the often *large size of graphical models*: Because of the extensive requirements in software design technology, the dimensions of graphical models can increase enormously. Moreover, Statecharts often are of great *complexity*: Because of the discrete nature of Statecharts, small changes not always have small effects. Beyond, Statecharts represent *interactive and distributed systems*: large collections of interconnected components usually involve interactive and concurrent processes. Therefore, potential errors can be very subtle and hard to locate for human developers.

3.2 Error Prevention

The approaches to error prevention in textual and visual languages face essentially the same problems. Due to this, we propose a common error prevention taxonomy and refine it in the following for Statecharts. Software error prevention in general encompasses a number of different techniques designed to identify programming flaws. As outlined in Figure 1a, we can basically distinguish between automated error prevention and human code review. As already pointed out, human code reviews are exceedingly time-consuming and often undependable in revealing errors. However, they may find conceptual problems that are impossible to detect automatically.



(a) Software Error Prevention in General and its Taxonomy.

(b) Taxonomy for Style Checking in Statecharts as a Refinement of General Software Error Prevention.

Fig. 1: Classification of Software Error Prevention.

Automated error prevention is commonly separated into dynamic and static methods. *Dynamic testing* performs code evaluation while executing the program and attempts to detect deviations from expected behavior: *Static code analysis*, on the other hand, performs an analysis of computer software without actual execution of programs, but by assessing source or binary files to identify potential defects. While dynamic testing requires executable code, static methods can be applied much earlier in the development process. Static code analysis covers aspects ranging from the behavior of individual statements and declarations to the complete source code of a program. Use of the information obtained from the analysis varies from highlighting possible coding errors to formal methods that mathematically prove properties about a given program, *e. g.*, that its behavior matches that of its specification, commonly known as *model checking*.

Style checking, another aspect of static code analysis, is concerned with layout style, *i. e.*, common appearance, as well as syntactic and semantic style. The latter two are often collectively referred to as robustness analysis (see below). Style checking always requires the syntactic and semantic correctness of the code. *Robustness analysis* refers to the objective of eliminating certain types of errors and enforcing sound engineering practices. Robustness rules limit the general range of a given modeling/programming language, as they are entirely independent of what is being designed.

In the general context of static code analysis, one must distinguish syntactic and semantic correctness on the one hand and style checking on the other hand. On this foundation, as a first step toward systematically devising an extensive style guide for Statecharts, the following taxonomy, depicted in Figure 1b, was laid down:

Syntactic Analysis: The enforcement of syntax-related rules does, in general, not necessitate knowledge of model semantics.

Readability (or layout style) aims at a graphical normal form, *e. g.*, transitions connect states in a clockwise direction, charts contain a limited number of states, *etc.*

Efficiency (or compactness, simplicity) emphasizes superfluous and redundant elements from the Statechart model.

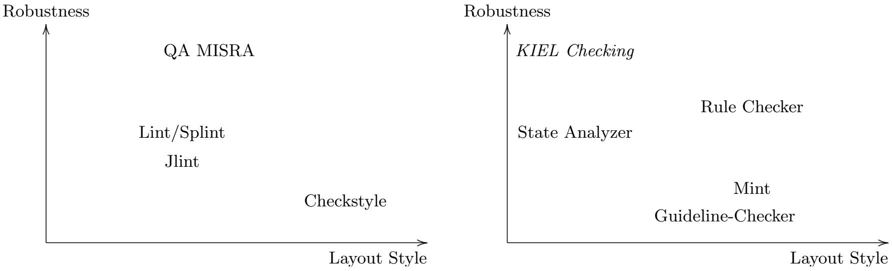
Syntactic Robustness aims at reducing errors due to inadvertence and enhancing maintainability.

Semantic Robustness: Deriving and enforcing semantic robustness rules requires knowledge of specific aspects of the model semantics. Exact analysis typically requires the use of formal verification tools.

3.3 Existing Style Guides and Applications

Style checking is based upon *style guides*. They constitute a set of design rules, concerning textual programming, respectively the modeling of Statecharts. Style guides provide general instructions on how to use languages. They are commonly provided as (in-)formal specifications, containing lists of rules. Style guides concern human languages, textual programming languages, as well as visual programming languages, such as Statecharts. They define a subset of usable elements. The informal as well as the formal specifications are primarily *operational instructions for humans*. These affect the programmed or modeled result. Beyond, formal style guides act as the configuration for *automated style checking*, *i. e.*, style checkers.

Since programming style often depends on the programming language, different coding standards and related code checking tools exist for different programming languages. Akin to coding standards, most code checking tools are programming language-specific. Available code checkers for C are *e. g.*, *Lint* [16], *LCLint* (aka. *Splint*) [17] and QA MISRA[10]; code checkers for Java are Jlint[11] and Checkstyle[12]. Figure 2a roughly classifies these code checkers according to their emphasis on layout style *vs.* robustness—a major distinction within style checking (see Section 3.2).



(a) Checking Tools for Textual Programming Languages.

(b) Checking Tools for Statecharts.

Fig. 2: Classification of Checking Tools according to their Emphasis of Layout Style *vs.* Robustness.

Statechart style checking is much less developed and less sophisticated as compared to style checking in textual computer programming. Nevertheless, when analyzing the dynamics of reactive systems, it is all the more important that models are designed according to approved rules. Therefore, several Statechart modeling tools, *e. g.*, *Stateflow* and *Statemate*, have been supplemented with a number of checks. Four representative checking tools—*Mint* and the *Guideline-Checker* related to *Stateflow*, the *State Analyzer* related to *Statemate*, and the *Rule Checker*—as well as our own robustness checker (see Section 5), are roughly classified according to their emphasis of layout style *vs.* robustness in Figure 2b.

The *Guideline-Checker* and the *State Analyzer* as well as Ricardo's *Mint* all address only a single Statechart dialect. *Mint*, the *Guideline-Checker*, and the *Rule Checker* merely perform graphical and—partly trivial—syntactic checks, but not profound semantic checks which require automated theorem proving as realized in the *State Analyzer*. However, semantic checks are particularly important since they eliminate possible non-trivial sources of error, which are very hard to discern for humans. The rules put forth in the next section aspire to fill this gap.

[10] `http://www.programmingresearch.com/`

[11] `http://jlint.sourceforge.net/`

[12] `http://checkstyle.sourceforge.net/`

4 Statechart Style Guide

Building on the aforementioned theoretical foundation, practical experience, and available prototypes, this section outlines a comprehensive Statechart style guide, striving for general applicability to Statechart dialects, within the limits of the UML State Machines specification. The rules presented below were formulated following the advice of Buck and Rau [18]: Clarity, Minimality, Consistency, Consensus, Flexibility, Adaptability, Stability and Testability.

As mentioned above (cf. Section 3.2), style guides for Statecharts can roughly be divided into two parts, namely syntactical analysis on the one hand and semantical analysis on the other hand. Syntactical analysis addresses the syntactical structure of Statecharts, such as layout, possible optimizations, and robustness problems. Therefore, in the context of syntactical rules, one basically has to focus on problems that deal with the relations of individual Statechart elements to each other. Furthermore, syntactical analysis opens up two fields of possible applications. One field analyses whether the syntactical relation of the elements used corresponds to the rules specified by a certain dialect (*i. e.*, syntactical correctness). Within the UML these kind of rules are called *well-formedness rule*s. The *well-formedness rule*s "[. . .] specify constraints over attributes and associations defined [with]in the [Statechart] meta model" [19, Section 2.3.2.2].

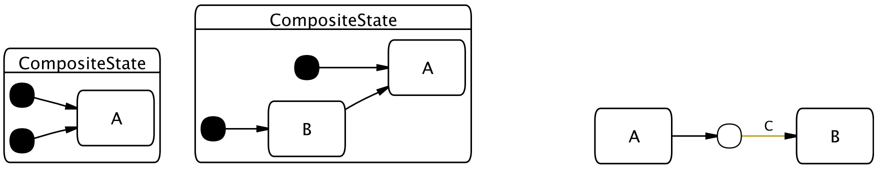
Nevertheless, locating problems from the part of syntactical correctness and syntactical robustness works the same way. Since Statecharts are directed graphs, one can use pattern matching here. If used for locating problems one would create a pattern that captures the problem.

In the following, we present the rules incorporated into our Statechart style guide. Following the proposed taxonomy (see Figure 1b), the rules are grouped in different sections. First of all, the rules dealing with the syntactical correctness, the *well-formedness rule*s, are presented. On that foundation, we extend the style guide by afterwards presenting the rules for syntactical robustness. Finally, the rules for semantical robustness are presented.

UML Well-formedness Rules

As mentioned above, syntactical correctness is mandatory for robustness. Therefore, it is necessary to check, whether a Statechart is syntactically correct or not. For most Statechart dialects, this is done within a dialect dependent modeling tool. But when dealing with UML State Machines, one has to manually make sure that the above mentioned *well-formedness rule*s are preserved as some UML tools do not check those rules at all. Within the UML, the *well-formedness rule*s themselves are described using OCL. Given a context of application and the constraint itself, problems are detected fairly easy. In the following, we present some examples for violations of the *well-formedness rule*s. Section 5 elaborates on the OCL implementation of the presented examples.

The rule *CompositeState-1* denotes that "a composite can have at most one initial vertex" [19, Section 2.12.3.1]. Detecting violations of this rule, as presented in Figure 3a (left-hand side), is done by a two-part pattern. One part contains a composite state with no initial vertex and the other part contains a composite state with one initial vertex. If neither part matches the composite state is known to have more than one initial vertex. Fixing problems detected by this check has to be done with great care because the intended behavior has to be carefully remodeled as Statecharts can include parts in which it is not clear what to do as depicted in Figure 3a (right-hand side). The rule *Transition-5* denotes that "Transitions outgoing pseudostates may not have a trigger" [19, Section 2.12.3.8]. The violation detection pattern may just contain a transition with the type of the source set to pseudostate and no trigger specified (see Figure 3b).



(a) Violations of Rule *CompositeState-1*.

(b) Violation of Rule *Transition-5*.

Fig. 3: Violations of *well-formedness Rules*.

Syntactical Robustness Rules

The style guide for Statecharts proposed in this paper aims at covering a wide range of dialects. Therefore, we extracted syntactical rules from various other style guides (cf. Section 2) that are applicable to different dialects. Furthermore, we formulated rules based on our own experience in Statechart modeling.

The rules presented below were adopted from Mutz [8, p. 144f]. All of them apply to the area of syntactical robustness and are dialect-independent.

MiracleStates: All states except the root state and the initial states must have at least one incoming transition. Figure 4a depicts the violation of this rule.

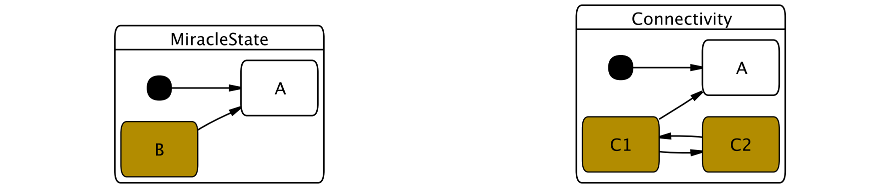
IsolatedStates: An even stronger version of *MiracleStates* is the check for isolated states. A state is isolated when it has neither incoming nor outgoing transitions.

EqualNames: Ensuring that all states are named differently simplifies the maintenance of a Statechart.

InitialState: Demanding that all regions respectively non-concurrent composite states contain one initial state greatly simplifies the understanding of the model. This rule should also be checked on dialects in which a region or non-concurrent composite state can be entered by an interlevel transition.

OrStateCount: Checking if all non-concurrent composite states contain more than one state delivers valuable hints for possible optimizations. Composite states that contain only one state can be subject to dialect independent optimizations and should be avoided from the beginning.

RegionStateCount: Closely related to *OrStateCount* this rule checks the number of states within a region of a concurrent composite state. Such regions can also be optimized and should be avoided for simplicity.



(a) Violation of the Rule *MiracleStates*. (b) Violation of the Rule *Connectivity*.

Fig. 4: Violation of Syntactical Robustness Rules.

From the Ford style guide the following rule was extracted as it is also applicable to dialects other than *Stateflow*.

DefaultFromJunction: When using connective junctions to model decisions one shall always add an outgoing transition with no label. The unlabeled transition is then the default transition. The default transition is provided so the control flow does not stop when the other conditions do not hold.

From our own experience in modeling with Statecharts the following rules were formulated.

TransitionLabels: Ensuring that all transitions are specified with a label makes the understanding of the model easier. This is especially important for dialects in which a default signal exists as it would be assigned invisibly to an unlabeled transition.

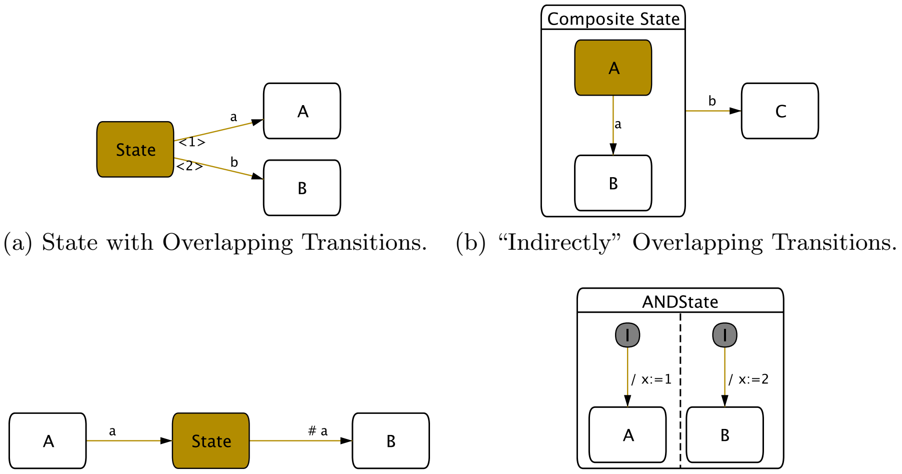
InterlevelTransitions: A Statechart should not contain interlevel transitions, *i. e.*, transitions bypassing level borders. The benefit is that understanding a Statechart without interlevel transitions is easier; especially novices tend to misunderstand the so expressed behavior, *e. g.*, the order of executed (entry) activities and the activation of parallel areas of execution.

Connectivity: Another aspect closely related to *MiracleStates* are states not connected by a sequence of transitions outgoing from any initial state. Such States are superfluous as they will never be entered while simulation. See Figure 4b where no path from the initial state to *C1* or *C2* exists. This rule extends the already mentioned *MiracleStates* as it also detects states that have incoming transitions and are still never entered as depicted in Figure 4b.

As mentioned above, locating a problem is fairly easy. However, resolving a found problem from the field of syntactic analysis can be more difficult. Depending on the context in which the problem is found and the problem itself, a different approach has to be used for each problem. Essentially, one can say that there is no general pattern applicable to all problems. Resolving found problems has two benefits. One benefit is that syntactical correctness of a Statechart will be achieved. This applies especially to the *well-formedness rules* of the UML. The more important benefit is, however, that the maintainability and the readability will increase enormously.

Semantic Robustness Rules

In line with the taxonomy presented in Figure 1b, we now turn to semantic robustness rules, addressing the model's behavior. As opposed to model checking, however, semantic robustness analysis is concerned with the behavior of individual statements and their interactions at a local level, *e. g.*, determinism and race-conditions. As, for the three rules presented below, transitions are considered pairwise, let $trans_1$ and $trans_2$ be the two transitions under investigation. The label of $trans_i$ is l_i, which consists of the predicates e_i (event expression) and c_i (condition expressions) as well as an action expression a_i, where $i \in \{1, 2\}$.



(a) State with Overlapping Transitions. (b) "Indirectly" Overlapping Transitions.

(c) *Dwelling* Violation. (d) Write/Write Race Condition.

Fig. 5: Application Examples of the Semantic Robustness Rules.

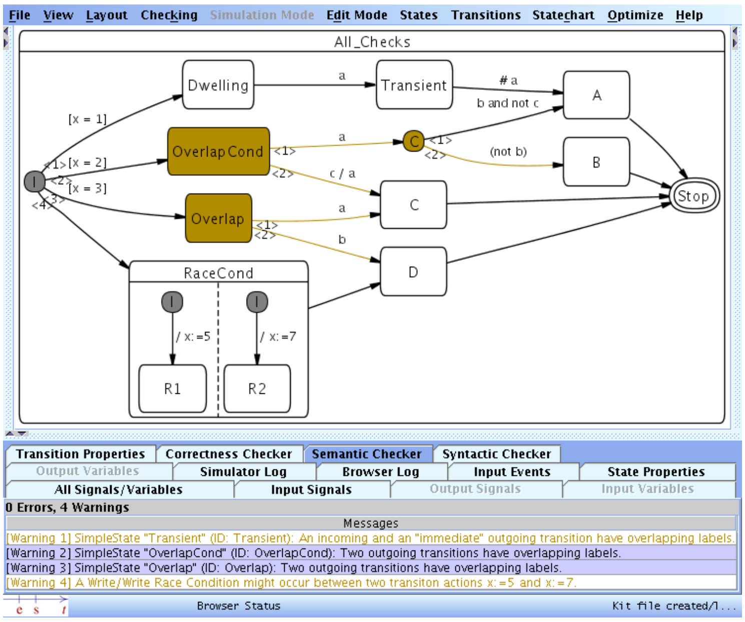
Transition Overlap: All transitions (directly or indirectly) outgoing from a state should have semantically disjoint predicates [20]. Ensuring this warrants that at most one transition is enabled at any time, *i. e.*, no transition shadowing can occur, leading to guaranteed deterministic behavior, independent of potential transition priorities. Figure 5a depicts a basic case of two transitions departing from a simple state. A *Transition Overlap* violation exists if e_1 and c_1 are not disjoint from e_2 and c_2. Such a violation may be eliminated by, *e. g.*, adding $\neg e_2$ and $\neg c_2$ to the predicates of $trans_1$, yielding $(e_1 \wedge \neg e_2)$ for the event expression and $(c_1 \wedge \neg c_2)$ for the condition expression. In addition to transitions departing directly from a state, transitions departing from an enclosing state may also be enabled (see Figure 5b). Overlaps are, however, resolved by transition priorities or hierarchy. Hence, this rule is primarily intended for Statechart dialects that do not provide a priority mechanism, such as *Statemate*.

Dwelling: The predicates of all incoming and outgoing transitions of a state should be pairwise disjoint or at least not completely overlapping [20]. This rule ensures that the system pauses at every state it reaches. A state in which the system cannot pause contradicts the concept of a *system state*. Careless use of *Esterel Studio*'s *immediate* flag, denoted by #, may lead to a *Dwelling* violation (see Figure 5c for an example). An immediate transition is evaluated in the same instant, in which its source state is reached; a non-immediate transition is not evaluated until the following instant.

Race Conditions: Concurrent writing or concurrent reading and writing of a variable should not exist in parallel states (cf. Figure 5d). Since race conditions are generally not detectable, we have chosen a conservative approximation. We detect a race condition in concurrent threads, if a variable is written in one thread and read or written in another. This rule, and the previous rule are aimed primarily at *Safe State Machine*s used in *Esterel Studio*.

5 The *KIEL* Modeling Environment

The *Kiel Integrated Environment for Layout* (*KIEL*) is a prototypical modeling environment that has been developed for the exploration of complex reactive system design [21]. As the name suggests, a central capability of *KIEL* is the automatic layout of graphical models. One can use *KIEL* to easily perform a layout of a given Statechart. However, the tool's main goal is to enhance the intuitive comprehension of the behavior of the System Under Development (SUD). While traditional Statechart development tools merely offer a static view of the SUD during simulation, in contrast, *KIEL* provides a simulation based on *dynamic focus-and-context* [21]. It employs a generic concept of Statecharts which can be adapted to specific notations and semantics, and it can import Statecharts that were created using other modeling tools. The currently supported dialects are those of *Esterel Studio*, *Stateflow*, and the UML via the XMI format, as, *e. g.*, generated by *ArgoUML*. *KIEL* further provides a structure-based editor to create Statecharts from scratch or to modify imported charts. A simulator is also part of the tool. The robustness checker, comprising checks for the rules presented above, has been integrated into *KIEL*. Figure 6 shows a screen-shot of *KIEL* as it checks particularly semantic robustness rules.



0 Errors, 4 Warnings

[Warning 1] SimpleState "Transient" (ID: Transient): An incoming and an "immediate" outgoing transition have overlapping labels.
[Warning 2] SimpleState "OverlapCond" (ID: OverlapCond): Two outgoing transitions have overlapping labels.
[Warning 3] SimpleState "Overlap" (ID: Overlap): Two outgoing transitions have overlapping labels.
[Warning 4] A Write/Write Race Condition might occur between two transiton actions x: =5 and x: =7.

Fig. 6: Screen-shot of *KIEL* Checking Robustness of a Statechart.

The Checking Plug-in

The checking plug-in of *KIEL* was designed to be very flexible in usage. All checks have been implemented independently. Via an user interface it is easily possible to manually select which checks to apply. It is further possible to define Statechart dialect-specific profiles containing different sets of rules. Depending on the model loaded into *KIEL*, the plug-in automatically decides which profile to apply.

The plug-in was developed to be easily extendable. The user can extend the rule set by either adding an appropriate OCL constraint for a syntactical check or by adding a new Java class for semantical checks. Depending on the seriousness of a detected problem, the robustness checker delivers two kinds of messages. (1) *Errors* in modeling are violations of rules that have to be addressed because further actions such as simulating the model is impossible. (2) *Warnings* indicate that a problem was found which does not need to be fixed immediately for simulation, *i. e.*, possible sources of errors or ambiguous constructs. In the following, an overview of the implementation of the aforementioned rules is presented.

We have chosen the OCL because, as stated by Mutz, it allows to formulate checks on a high level of abstraction, and neither knowledge of a programming language nor of the underlying data structure is needed [8]. The executive approach towards the evaluation of OCL is preferable to an interpretative approach as the former one proved to be more flexible and faster in execution time.

Therefore, we chose to use the Dresden OCL Toolkit version 1.3[13] discussed by Richters [10] to transform OCL constraints to Java.

Our approach for the checking framework contains the possibility of returning customized messages when a violation is found. Therefore, the OCL constraint is wrapped by additional information as Java code snippets. The union of OCL and Java code snippets we named *KIEL wrapped OCL* (KOCL). The developed *KOCL* to Java translator utilizes the Dresden OCL Toolkit which is supplied with the according meta model of the *KIEL* data structure. Figure 7 basically shows how the different parts of the *KOCL* files are processed. The workflow and the specified rules were described in detail elsewhere [22].

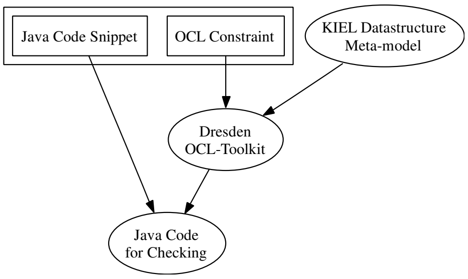


Fig. 7: Processing *KOCL* with *KIEL*.

As the framework is designed to handle rules formulated as OCL-constraints we have implemented the rules elaborated above (cf. Section 4). Most of the *well-formedness rul*es were specified in *KOCL*. The rules not specified in *KOCL* deal partly with features of UML diagrams. As the *KIEL* project so far is focused on simulating and modifying Statecharts only, the representations of classes and packages was left out for the sake of simplicity. Therefore, *e. g.*, rule StateMachine number 1 which states that "a State Machine is aggregated within either a classifier or a behavioral feature" from the UML specification was left out.

We will not present all of the transfered rules in detail. The example presented in the following gives an overview about how the additional information is capsuled within *KOCL* files. A relatively simple example is Rule *CompositeState-1* (cf. Section 4) as specified in Figure 8a. The OCL constraint states that the set `subvertex` of a composite state can contain at most one pseudostate of kind `#initial`. The Dot notation is used to access members of a class. An arrow ("`->`") is used to access properties or functions on sets.

The rule specified in *KOCL* is presented in Figure 8b. The separation of the message declaration, the constraint definition and the specification of the returning message is clearly seen in this example. The `declarations` part (lines 2–4) is designed to hold more than one message. The `fails` part (line 10) specifies

[13] `http://dresden-ocl.sourceforge.net/`

which message to return if a violation of the constraint is found. It is even possible to return different messages (if defined) depending on the context in the `fails` part by simply using a common `if-then-else`-statement. Due to the meta model the constraint itself (lines 7–9) is even shorter than specified in the UML.

```
self.subvertex->select(
  v| v.oclIsKindOf(Pseudostate))->
select(
  p:Pseudostate| p.kind = #initial)->
size <= 1
```

(a) The OCL Representation.

```
 1  rule UML13CompositeStateRule1 {
 2    declarations {
 3      message "A composite state can have
 4            at most one initial vertex.";}
 5    constraint {
 6      context ORState or Region;
 7      "self.subnodes->select(
 8         v| v.oclIsTypeOf(InitialState))->
 9         size <= 1";}
10    fails {message;}
11  }
```

(b) The *KOCL* Representation.

Fig. 8: The Rule *CompositeState-1.*

As mentioned before, syntactic analysis is not the only field for which an automated checking framework for Statecharts is beneficial. The presented semantic rules can also be checked automatically. Although OCL is of great benefit in specifying and implementing robustness checks regarding the syntax of Statecharts, semantic analyses are generally beyond its scope because checking a Statechart with respect to these rules typically requires extensive knowledge of the model semantics. The *Transition Overlap* rule, the *Dwelling* rule, and the *Race Conditions* rule (see Section 4) cannot be specified using OCL constraints. Our framework still allows to incorporate such checks; for this purpose Java code is needed to formulate theorem-proving queries and sending them to an outside tool for analysis.

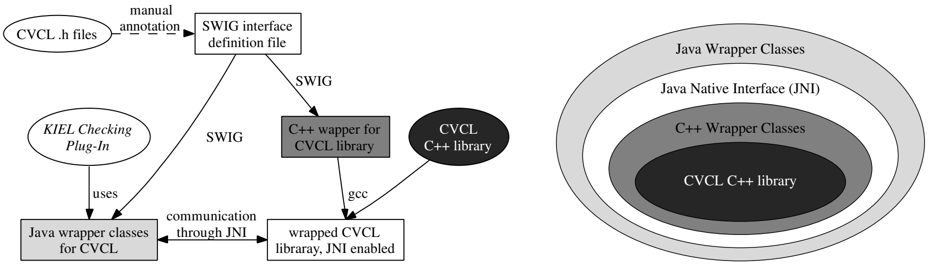
To perform the semantic robustness checks, a *satisfiability modulo theories* (SMT)[14] solver is needed. SMT problems are a variation of *automated theorem proving* [23], which in turn is part of automated reasoning. After an evaluation of available SMT solvers [24], CVC Lite [25] was chosen. Here, in order to determine whether, *e. g.*, two transitions $trans_1$ and $trans_2$ (cf. Section 4) have overlapping labels, satisfiability of the formula

$$\Big((e_1 \wedge c_1) \wedge (e_2 \wedge c_2)\Big)$$

must be decided. Unsatisfiability implies that the predicates of $trans_1$ and $trans_2$ are disjoint. Such SMT problems are generally decidable as long as they contain only addition but no multiplication of variables.

[14] `http://combination.cs.uiowa.edu/smtlib/`

Further, the *Simplified Wrapper and Interface Generator* (*SWIG*) [26] was employed to generate wrappers and interface files for CVC Lite, enabling its immediate use from within Java. Here, the Java and C++ JNI wrappers are produced from CVC Lite's annotated C++ header files, as shown in Figure 9.


CVCL .h files
manual annotation
SWIG interface definition file
SWIG
SWIG
KIEL Checking Plug-In
C++ wapper for CVCL library
CVCL C++ library
uses
gcc
communication through JNI
Java wrapper classes for CVCL
wrapped CVCL libraray, JNI enabled
Java Wrapper Classes
Java Native Interface (JNI)
C++ Wrapper Classes
CVCL C++ library

(a) Outline of the SWIG Workflow, Including the Link-Up of the CVC Lite Library to the *KIEL Checking* Plug-In.

(b) Conceptual Diagram of the Hierarchical Composition of Wrapper Layers around the CVC Lite Library.

Fig. 9: Interfacing of *KIEL* and the CVC Lite Library via JNI and SWIG.

6 Experimental Results

Finally, we show the application of the checking framework on a well known example, the wristwatch presented by Harel [1]. As this example is well-established we did not expect to detect real modeling errors; our focus was to quantitatively asses the efficiency of our checking mechanism. We remodeled the wristwatch with ArgoUML which imposed some restrictions. *E. g.*, some transitions perform indexing over multiple states, which was replaced by according conditional constructs. However, the final model retains the originally modeled behavior. So far, the final model contains 120 transitions and 108 states.

The results from benchmarking are presented in Table 1. The number of returned hints and the run-time of each check are presented. The checking times were measured on a PC with GNU/Linux OS, a 2.6 GHz AMD Athlon 64 processor and 2 GB of RAM.

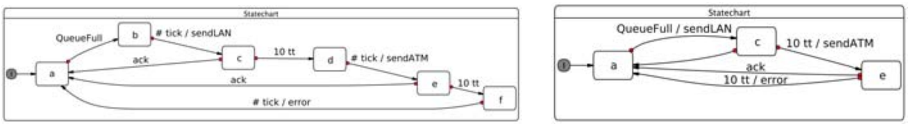
The application of the *well-formedness rul*es consumed the least time of all parts. Roughly 20 milli-seconds were needed to check those rules on the chart. Except for the check *EqualNames* the syntactical robustness checks roughly take twice as much time as the *well-formedness rul*es. The check *EqualNames* has a quadratic complexity in the number of states. This is caused by limitations of the OCL—all states have to be compared to the currently handled state. In comparison to the checks dealing with syntactical robustness, except *EqualNames*, the checks for semantical robustness take about 400 milli-seconds. Here, the check

Table 1: Experimental Results of Checking the Wristwatch Example.

Checks	**Hints**	**Time [ms]**
well-formedness checks (total)	**0**	**20**
InterlevelTransition	17	14
Connectivity	7	2
EqualNames	33	587
InitialStateCount	7	1
TransitionLabels	6	9
IsolatedStates	1	4
syntactical checks (total)	**71**	**617**
Transition Overlap	598	352
Dwelling	0	2
Race Conditions	0	1
semantical checks (total)	**598**	**355**
total	**669**	**992**

Transition Overlap returns an enormous number of hints compared to the total number of transitions. This is due to the fact that almost no transition was designed with an opposing predicate of another outgoing transition.

As another example, the application of the framework on the Statechart presented in Figure 10a delivered the hint that violations of the rule *Dwelling* are present. Especially novices tend to produce unnecessarily large models with needless states, for example by splitting trigger and effect into separate transitions. Figure 10b shows a possible way how the violation can be fixed. Because the Statechart is rather small, all checks were applied in about 3 milli-seconds.



(a) A Statechart with Unnecessary, Transient States.

(b) Statechart after Removing the Transient States b, d, and f.

Fig. 10: Example for Removing Transient States.

7 Assessment

We gained the following results during the work with the framework. Not surprisingly, the time needed for specifying rules differs significantly depending on the complexity of the problem. The fairly simple *well-formedness rule*s from the UML were specified in *KOCL* in a rather short amount of time. All in all, it took less than one hour to specify them. The more sophisticated rules regarding problems from the field of syntactical robustness took not much longer, as the OCL proved to be an easy to apply language for these kind of problems, too. The time needed for specifying those rules varies between two minutes and half an hour per rule. The semantic robustness rules turned out to be the most demanding. Roughly two weeks were needed altogether for the implementation of the three rules presented. The main aspect of this task was to extract the needed data and afterwards to transform the data from the Statechart to the input language of the theorem prover.

The *well-formedness rule*s do not necessarily improve the quality of Statecharts in the sense of robustness. Those rules apply to the field of syntactical correctness only. Nevertheless, these rules are needed before any further checking can be applied to a Statechart, because the robustness checks rely on the correct syntax.

Syntactical robustness rules, however, focus on more intricate problems, but not as sophisticated as the rules dealing with the semantical robustness. Nevertheless, the information gained by applying the checks is worth it. The information delivers sources for possible optimizations that lead to a better understanding of the checked Statechart. *E.g.*, the readability of charts significantly improves if all states are labeled with different names. Furthermore, the tests for *Connectivity* and for *MiracleStates* may detect design flaws that may lead to misbehavior of the modeled system. Therefore these problems should always be corrected to fix the model and also to increase the maintainability.

The *Transition Overlap* and *Dwelling* rules certainly improve the structural clarity of Statecharts, as all behavior is diagrammed explicitly. Especially in a non-deterministic dialect such as *Statemate*, the introduction of determinism greatly eases model comprehension. The *Race Conditions* rule, on the other hand, might be too restrictive in real life. If applied, though, it leads to immense structural improvements as potential race conditions in far apart regions of a Statechart are eliminated *a priori*.

Finally, there is a trade-off between semantic robustness and minimality of Statecharts. *E. g.*, eliminating a *Transition Overlap* or *Dwelling* violation by adding the negation of the predicates of one transition to the predicates of the other transition, as suggested above, constitutes an infringement of the *write things once* principle of modeling [27].

In summary, on the one hand one can say that evaluating OCL statements as specified by the *well-formedness rule*s turned out to be a task very fast done. On the other hand one has to say that for statements of greater complexity the evaluation of OCL—as in rule *EqualNames*—can be much more time consuming.

8 Conclusion and Further Work

As the failure of safety-critical systems can have severe consequences, error prevention in the model-driven system development of such systems is vital. We have outlined an approach to make the model-driven system development with Statecharts less error prone, and have presented a general Statechart style guide that is not restricted to a single dialect. We implemented a flexible robustness checking framework within the *KIEL* modeling tool. The hints returned by our checking framework do not necessarily indicate errors; this typically still requires application-specific knowledge. However, as has been observed in earlier work, adhering to the robustness rules reduces the chance for errors. Beyond that, they serve to improve the readability and maintainability of a system.

Our framework permits to express robustness rules directly with the well-established OCL formalism, which facilitates an abstract rule formulation and allows to directly incorporate existing OCL rule sets. Our transformative approach for the evaluation of OCL statements has turned out superior to earlier, interpretative approaches, and the expressiveness of the OCL has been sufficient for most of our checks. However, the framework also allows to implement complex semantic checks in Java directly, which we have used to incorporate an off-the-shelf theorem prover. Our framework has been practically validated for the checking of UML State Machines; however, the framework could easily be adapted to other commercial Statechart modeling tools as well; provided that an appropriate import functionality exists.

Beyond the experimental results presented in this paper, we intend to utilize the *KIEL* checking framework to perform a systematic study of the effectiveness of robustness checking, both for novice users and experienced modelers. Furthermore, we plan to implement support for the recently published version 2.0 of the OCL, and to incorporate further rules into our checking framework.

References

[1] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (1987) 231–274

[2] Object Management Group: Unified Modeling Language: Superstructure, Version 2.0 (2005)

[3] Parnas, D.L.: Some theorems we should prove. In: HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications, London, UK, Springer-Verlag (1994) 155–162

[4] Object Management Group: (Unified Modeling Lanugage—UML Resource Page) `http://www.uml.org`.

[5] Motor Industry Software Reliability Association (MISRA): MISRA-C:2004. Guidelines for the Use of the C Language in Critical Systems. Motor Industry Research Association (MIRA), Nuneaton CV10 0TU, UK (2004)

[6] Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a "safe" subset of simulink/stateflow into lustre. Technical Report 2004-16, Verimag, Centre Équation, 38610 Gières (2004)

[7] Kreppold, T.: Modellierung mit Statemate MAGNUM und Rhapsody in Micro C. Berner & Mattner Systemtechnik GmbH, Otto-Hahn-Str. 34, 85521 Ottobrunn, Germany, Dok.-Nr.: BMS/QM/RL/STM, Version 1.4 (2001)

[8] Mutz, M.: Eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich. Dissertation, Technische Universität Braunschweig (2005)

[9] Pap, Z., Majzik, I., Pataricza, A.: Checking general safety criteria on UML statecharts. Lecture Notes in Computer Science **2187** (2001)

[10] Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, University of Bremen (2001)

[11] Mutz, M., Huhn, M.: Automated statechart analysis for user-defined design rules. Technical report, Technische Universität Braunschweig (2003)

[12] Moutos, M., Korn, A., Fisel, C.: Guideline-Checker. Studienarbeit, University of Applied Sciences in Esslingen (2000)

[13] Scheidler, C.: Systems Engineering for Time Triggered Architectures. SETTA Consortium (2002) Deliverable D7.3 – Final Document.

[14] Heitmeyer, C., Jeffords, R., Labaw, B.: Automated Consistency Checking of Requirements Specifications. ACM Transactions on Software Engineering and Methodology **5**(3) (1996) 231–261

[15] Heimdahl, M.P.E., Leveson, N.G.: Completeness and Consistency in Hierarchical State-Based Requirements. Software Engineering **22**(6) (1996) 363–377

[16] Johnson, S.C.: Lint, a C program checker. In Thompson, K., Ritchie, D.M., eds.: UNIX Programmer's Manual. Seventh edn. Bell Laboratories (1979)

[17] Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **19**(1) (2002) 42–51

[18] Buck, D., Rau, A.: On Modelling Guidelines: Flowchart Patterns for STATEFLOW. Softwaretechnik-Trends **21**(2) (2001) 7–12

[19] Object Management Group: Unified Modeling Language (UML) 1.3 specification (2000) `http://www.omg.org/cgi-bin/apps/doc?formal/00-03-01.pdf`.

[20] Kossowan, K.: Automatisierte überprüfung semantischer modellierungsrichtlinien für statecharts. Diplomarbeit, Technische Universität Berlin (2000)

[21] Prochnow, S., von Hanxleden, R.: Comfortable Modeling of Complex Reactive Systems. In: Proceedings of Design, Automation and Test in Europe (DATE'06), Munich (2006)

[22] Bell, K.: Überprüfung der Syntaktischen Robustheit von Statecharts auf der Basis von OCL. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik (2006) unpublished.

[23] Gallier, J.H.: Logic for Computer Science: Foundations of Automatic Theorem Proving. Revised On-Line Version (2003), Philadelphia, PA (2003)

[24] Schaefer, G.: Statechart Style Checking – Automated Semantic Robustness Analysis of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik (2006)

[25] Barrett, C.W., Berezin, S.: CVC Lite: A new implementation of the Cooperating Validity Checker Category B. In Alur, R., Peled, D.A., eds.: Proceedings of Computer Aided Verification: 16th International Conference, CAV 2004, Boston. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 515–518

[26] Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop. (1996) 129–139

[27] Berry, G.: The Foundations of Esterel. Proof, Language and Interaction: Essays in Honour of Robin Milner (2000) Editors: G. Plotkin, C. Stirling and M. Tofte.

Time Exceptions in Sequence Diagrams

Oddleif Halvorsen[1], Ragnhild Kobro Runde[2], Øystein Haugen[2]

[1] Software Innovation
[2] University of Oslo
{oddleif|ragnhilk|oysteinh}@ifi.uio.no

Abstract. UML sequence diagrams partially describe a system. In this paper we show how the description may be augmented with exceptions triggered by the violation of timing constraints. We compare our approach to those of the UML 2 simple time model, the UML Testing Profile and the UML profile for Scheduling, Time and Performance. We give a formal definition of time exceptions in sequence diagrams and show that the concepts are compositional. An ATM example is used to explain and motivate the concepts.

1 Introduction

UML sequence diagrams [8] are a useful vehicle for specifying communication between different parts of the system (e.g. components or objects). A sequence diagram specifies a set of positive traces, and a set of negative traces. A trace is a sequence of events, representing a system run. The positive traces represent legal behaviors that the system may exhibit, while the negative traces represent illegal behaviors that the system should not exhibit.

Timing information may be included in the diagram as constraints. These constraints may refer to either absolute time points (e.g. the timing of single events) or durations (e.g. the time between two events). The described behavior is negative if one or more time constraints are violated.

In practice, it may often be impossible to ensure that a time constraint is never violated, for instance when the constrained behavior involves communication with the environment. Usually, a sequence diagram does not describe what should happen in these exceptional cases. In this paper we demonstrate how the specification may be made more complete by augmenting the sequence diagram with exceptions that handle the violation of time constraints. The ideas behind our approach originate from [1], which treats exceptions triggered by wrong or missing data values in the messages.

Time violations are exceptional situations that are not supposed to happen very often. Modeling violation of time constraints as exceptions rather than using the alt-operator for specifying alternative behaviors, has the advantage that

- specifying the exceptional behavior separately from ordinary/expected behavior simplifies the diagrams and increases the readability,
- after having specified the normal behavior, exceptional behavior can easily be added in separate exception diagrams.

A single event may violate a time constraint in three different ways:

1. The event occurs too early.
2. The event occurs too late.
3. The event does not occur at all.

All three situations will result in an exception, but the exact exception handling to be performed will typically be very different depending on the nature of the violation. In this paper we focus on the last case, where an event has not occurred within the given time limit and we therefore assume that it will not occur at all. If the event for some reason occurs at some later point it should be treated as another exception.

2 Background

In this section we motivate our work by presenting state of the art regarding timing constraints in UML. The main conclusion may be summarized as follows: Both the UML 2.1 simple time model (Section 2.1) and the UML profile for Schedulability, Performance and Time (Section 2.2) introduce concepts and notations for defining time constraints, but do not consider what should happen in case of violations. TimedSTAIRS (Section 2.3) distinguishes between the reception and the consumption of a message, but as it is based on UML 2.1 simple time model, TimedSTAIRS does not consider violations either. The default concept of UML Testing Profile (Section 2.4) and our previous work on exception handling (Section 2.5) consider violation of constraints, but mainly regarding wrong or missing data values, and not time constraint violations.

2.1 The UML 2.1 Simple Time Model

UML 2.1 [8] includes a simple time model intended to define a number of concepts relating to timing constraints. In general the semantics of the timing constraints follow the general interpretation of constraints in UML: "A Constraint represents additional semantic information attached to the constrained elements. A constraint is an assertion that indicates a restriction that must be satisfied by a correct design of the system."

Furthermore the timing constraints always refer to a range of values, an interval. The concept IntervalConstraint is defined by: "The semantics of an IntervalConstraint is inherited from Constraint. All traces where the constraints are violated are negative traces (i.e., if they occur in practice the system has failed)." Some notation is introduced to define these IntervalConstraints shown by the example in Fig. 1.

We notice that UML 2.1 only states that when the constraints are violated the system is in error. Exceptions triggered by time constraint violations are not considered.

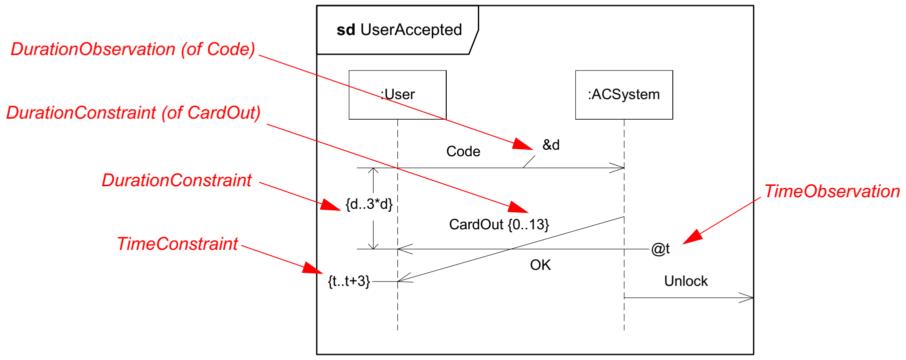


Figure 13.15 - DurationConstraint and other time-related concepts

Fig. 1. UML 2.1 Timing constraints example

2.2 UML Profile for Schedulability, Performance and Time

The UML profile for Schedulability, Performance and Time Specification [6] is a profile based on UML 1.4 [5] describing in great detail concepts relating to timely matters. The profile, hereafter referred to as SPT, is a profile based on UML 1.4 and therefore a profile that will have to be updated to UML 2.0. There is now ongoing work to upgrade the realtime profile under the name MARTE.

SPT introduces a large number of concepts. They represent most often properties of behavioral units needed to schedule these units and to analyze their performance.

Exceptions are not mentioned at all, but the following are some notes on the time mechanisms.

By introducing concepts that allow to define "timing marks", it is possible to describe constraints on these timing marks, and in principle express time and duration constraints similar to what is the case with UML 2 simple time model.

SPT goes no longer than the simple time model when it comes to constraints. Of course SPT allows constraints to be expressed on a large number of properties having been declared on behavioral units, but it never considers what happens if the constraint is not met. Implicitly this means that if the constraint is not met, the system is in complete failure.

In Fig. 2 we show an example of a fairly crowded diagram with a number of constraints on properties that go beyond pure UML.

2.3 TimedSTAIRS

TimedSTAIRS [3] is an approach to the compositional development of timed sequence diagrams. With time constraints, it is important to know whether a given constraint applies to the reception or the consumption of the message.

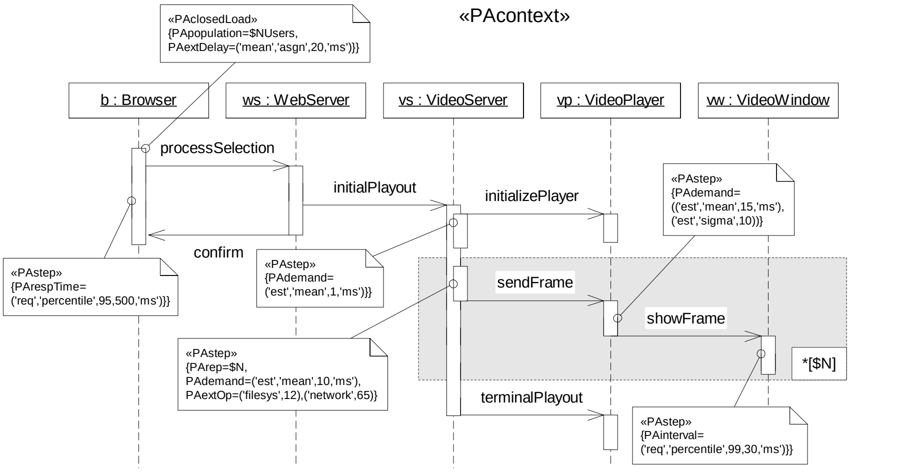


Figure 7-8 Sequence diagram of web video application with performance annotations (partial set)

Fig. 2. SPT diagram with constraints

Hence, in [3] we argued for a three-event semantics of timed sequence diagrams. The example given is reproduced here in Fig 3.

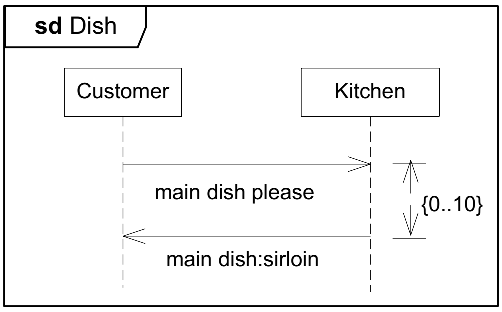


Fig. 3. Timing constraint as presented in [3]

Here, the time constraint specifies that the kitchen should not use more than ten time units from it receives an order until it is served. However, does the time constraint apply to when the order is placed in the kitchen's order-queue, or from when the chef actually reads the order and starts working on it? In this case, the customer probably wants the time constraint to apply to the kitchen receiving the message, while in other situations the constraint will apply to the consumption. In order to make a graphical distinction between reception and consumption, [3] uses a double arrow for reception, and the standard single arrow for consumption. We will follow this convention in our examples later in this paper. If only the consumption event is present in the diagram, the reception event is taken implicitly, while if only the reception event is present, the implicit consumption event may or may not take place.

In TimedSTAIRS, the semantics of a diagram like the one in Fig. 3 is a set of positive (i.e. legal) behaviors and a set of negative (i.e. illegal) behaviors. In Fig. 3, the positive traces are all traces where the customer asks for main dish, and the kitchen serves sirloin within ten time units. The negative traces are all traces where the customer asks for main dish, and the kitchen uses more than ten minutes before serving sirloin. All traces that are not described in the diagram (e.g. if the customer asks for dessert, or if the kitchen serves chicken instead of sirloin, regardless of the time it takes) are referred to as inconclusive. These may later be supplemented (i.e. added) to the specification as either positive or negative, in order to make the specification more complete.

2.4 UML Testing Profile — default concept

The U2TP (UML Testing Profile) [7] introduces the notion of Defaults that aims to define additional behavior when a constraint is broken.

We will go through an example found in the standard, and present here the semantics of defaults in sequence diagrams (Interactions) quoted from the U2TP standard:

> For defaults that are described on Interactions, the semantics is given as an algorithm that combines the traces of the default behavior with the traces of the main description:
>
> Assume that there is a main description of an interaction fragment. Its semantics can be calculated as a set of traces. This set of traces is projected onto the test component with the default by removing all event occurrences of other lifelines from the traces. The result is a set of traces only involving event occurrences of the test component. This is called the main description. Every trace in this set can be split in three parts: a head, a trigger, and a tail. The trigger is normally a receiving event occurrence. One particular trace can therefore be constructed in parts in several ways. The default is a behavior and is therefore also a set of traces, each of which can be divided in two parts: a trigger and a tail. For every trace in the main description, construct more traces by concatenating main-head with every default trace provided that main-trigger is different from default-trigger. Retain the information on a trigger that it was originally a default-trigger by a flag. Finally filter out all traces starting with main-head+trigger-with-default-flag if there is another trace in the set starting with main-head+main-trigger and the main-trigger is equal to the trigger-with-default-flag. This makes up the set of traces for the interaction fragment with associated default.

Fig. 4 presents a very small example based on an excerpt of one of the diagrams given in the standard.

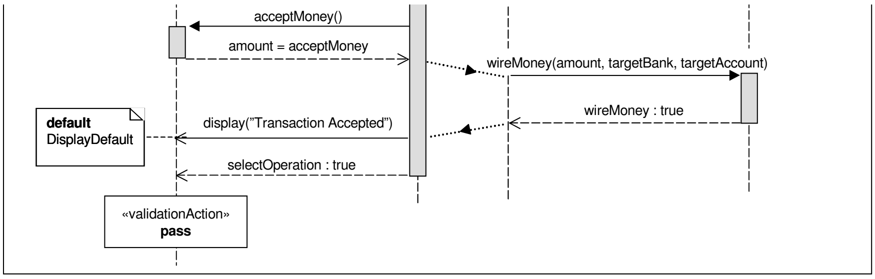


Fig. 4. U2TP default example (from Figure 6.29 of the U2TP standard)

Disregarding the default (exception) we have the following traces: $\{\langle !cam, ?cam, !ram, ?ram, !cwM, ?cwM, !rwM, ?rwM, !di(T), ?di(T), !rsO, ?rsO, pass\rangle, \langle !cam, ?cam, !ram, ?ram, !cwM, ?cwM, !rwM, ?rwM, !di(T), !rsO, ?di(T), ?rsO, pass\rangle\}$ where ! represents call or send event and ? represent the reception event. The default applies to lifeline hwe (the leftmost lifeline in Fig. 4) and the projection of the main traces onto this lifeline gives the trace $\{\langle ?cam, !ram, ?di(T), ?rsO, pass\rangle\}$ which can easily be seen from the diagram. If we consider only the DisplayDefault as the exception, the only possible main-trigger is $?di(T)$ and we have for the single trace of the projected set: main-head: $\langle ?cam, !ram\rangle$ main-trigger: $\langle ?di(T)\rangle$ main-tail: $\langle ?rsO, pass\rangle$.

In Fig. 5 we look at the definition of DisplayDefault.

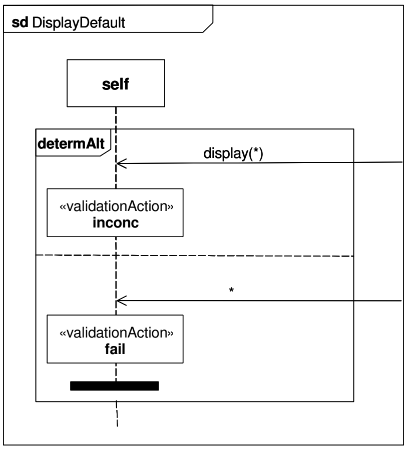


Figure 6.30 - Default for individual message reception

Fig. 5. U2TP default definition

The semantics for DisplayDefault is $\{\langle ?di(*), inconc\rangle, \langle ?*, fail\rangle\}$ where the * is the wildcard and denotes a range of values. The first one is any string other than T and the second asterisk is any signal other than di. The first event on either trace is their default-trigger and the second event is the default-tail.

The resulting set of event traces for hwe after applying the transformation described above, is the following: $\{\langle ?cam, !ram, ?di(T), ?rsO, pass\rangle,$ $\langle ?cam, !ram, ?di(*), inconc\rangle, \langle ?cam, !ram, ?*, fail\rangle\}$ This is reasonable and what we wanted. To get the total set of traces the projections on all lifelines must be combined in a parallel combination.

U2TP says little about the semantics of defaults triggered by the violation of time constraints. The idea behind the defaults on different levels is that even the notoriously partial interactions are made complete and actually describing all behaviors. But the U2TP definition is not adequately precise in this matter and there are no convincing examples given to explain what happens when a time constraint is violated.

2.5 Proposed notation for exceptions in sequence diagrams

In [1] we introduce notation for exceptions in sequence diagrams. The constraints that are violated are always on data values at the event associated with the exception. Violation of time constraints is not considered.

The semantics of the behavior including the exceptions are again given by a transformation procedure quite similar to that of U2TP. The idea is again that supplementing traces are defined in the exception starting from the prefix of traces leading up to a triggering event.

The other novelty of our approach in [1] is that it suggests a scheme of dynamic gate matching that makes it possible to define exceptions independently. That idea is orthogonal to what we try to convey in this paper.

3 Time exceptions in the ATM example

In this section we shall through an example with an Automatic Teller Machine (ATM) show how time exceptions supplement the description and make the specification more complete and comprehensive without losing sight of the normal scenarios. The ATM example is based upon the case from [1].

We will start by introducing what we call the normal flow for the ATM. The normal flow refers to a happy day scenario or what we would like to see the system do. The normal flow is in this paper not the most important part, while the exceptions are, and hence this introduction will be brief.

3.1 The normal flow

As our example, we will show the use of an ATM to withdraw money. Fig. 6 shows how the ATM interacts with users and banks.

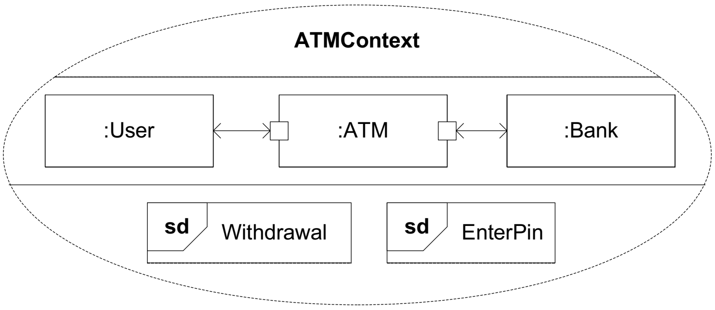


Fig. 6. The ATM context

The context has not specified any cardinality, but intuitively we know that one ATM only serves one user at a time, and that with a given card this user can only withdraw money from one single bank.

Withdrawal in Fig 7 specifies that the user is expected to insert a card, and enter a four digit Personal Identification Number (PIN) whereas the ATM is to send the pin to the bank for validation. While the bank is validating the pin, the user is asked to enter the amount to withdraw. When a valid pin is given, the bank will return OK. Then the ATM withdraws the money from the account and gives it to the user together with the card.

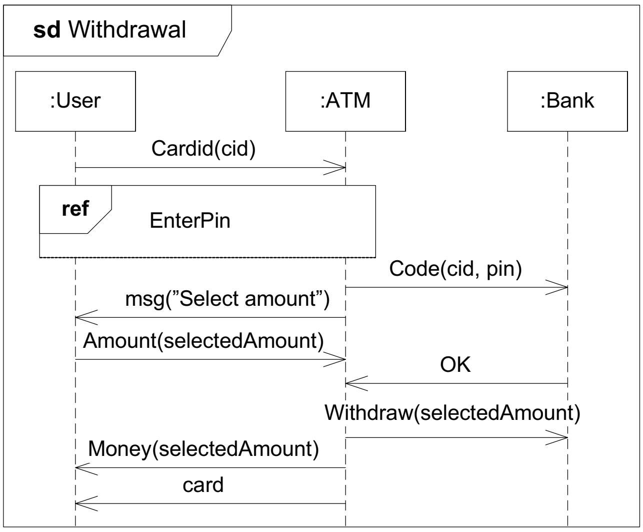


Fig. 7. Specification of withdrawal

EnterPin in Fig. 8 specifies how the user is to give the ATM the four digit PIN. It is here very important not to interpret an interaction use (here: referring EnterPin) as a method call like for example in Java. An interaction use is not a method call, but rather an inclusion of a fragment equal to the referred sequence diagram.

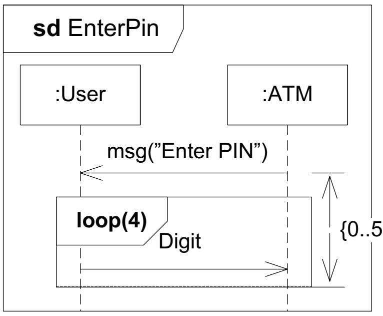


Fig. 8. Specification of entering a PIN

When you perform a method call in Java a new stack frame is added to the call stack. This means that there is a difference between keeping all the code in one method and splitting it into several methods. Sequence diagrams have no call stack, and hence there is no difference between adding an interaction use and including the traces for it directly in the sequence diagram.

What we have shown so far is the normal flow of control, which is what we want the system to do most of the time. On the other hand this specification is not very robust, and cannot serve as a complete specification for implementation. What may go wrong is that the user enters a wrong PIN, the ATM is out of money, the user's account is empty or the ATM loses contact with the bank.

These are just some of the more important exceptions one needs to handle to make a robust specification. What we need is some way to describe the possible exceptions for this specification.

Although we have stated that one needs to handle exceptions, sequence diagrams are still partial description. By that we mean that sequence diagrams are not supposed to cover every possible trace. What we aim to do is to make them more complete, in relation to the important parts of the system. Another goal is to make a clearer separation of what's normal and what's exceptional in sequence diagrams. This in turn should increase readability.

3.2 Applying time exceptions to the ATM

Sequence diagrams are often filled with various constraints, but they seldom say much about what to do if a constraint breaks. Hence the system has completely failed if a constraint is broken. In most situations this is inadequate.

In order to make the specification more robust, we will add time exceptions to the ATM case.

In the ATM case a possible time exception may be that the user for some reason had to run before completing the transaction, or that the bank uses too long time to validate the given PIN. These are both exceptions that have their basis in time as the ATM needs some perception of time to decide whether or not it has gone too long time since the last input.

When it comes to violations of time constraints we have three possible scenarios regarding events. The event may happen:

1. too early
2. too late
3. never

As explained in the introduction, in this paper we focus on the last case. By that we mean that if an event has not occurred within the specified constraint we assume it to never happen. If the event for some reason occurs after the constraint was violated it should be treated as another exception.

When it comes to the evaluation of time constraints we are building the semantics upon timestamps, whereas we assume that the running system performs some kind of surveillance of the system, in order to evaluate the constraints. Intuitively, this means that we consider time constraints to conceptually work like alarm clocks. If the associated event is too late the alarm goes off and the exception handler is triggered.

3.3 Time exceptions in EnterPin

We will start to present the notation by applying a time exception to the EnterPin diagram. A possible exception is that the user enters less than four digits or that the digits for some reason is not received by the ATM. If we don't handle this, the ATM will not be ready for use when the next user arrives.

In order to handle this, we need a way to assert that the user has really left, and if so we need to take the card from the card reader and store it some place safe before canceling the user's session.

This is shown in Fig. 9. We have added a time constraint stating that if the ATM has not received all the digits within the specified time, the exception UserLeftCard will fire.

The time constraint itself is initialized on the send event on msg, and attached on the bottom of the loop fragment. Attaching it to the bottom of the loop fragment indicates that the time constraint is to hold for the last message, and hence all the preceding ones as well. Notice that the exception is attached to the end of the time constraint, since the possible exceptions are to fire upon a broken time constraint.

Fig. 10 shows how the UserLeftCard exception is handled. In the case that the user leaves the ATM before proper completion of the service, the ATM sends a message stating that the service was canceled and then specifies that we are to terminate the service. By stating terminate we mean that the service, withdrawal of money, is to terminate — not the whole ATM. We will elaborate more on that in the next section, but for now interpret terminate as the termination of the overall service.

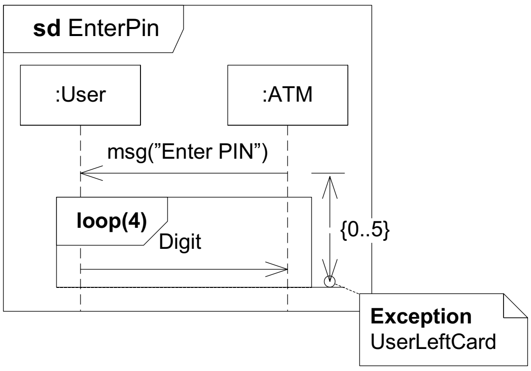


Fig. 9. EnterPIN with time exception

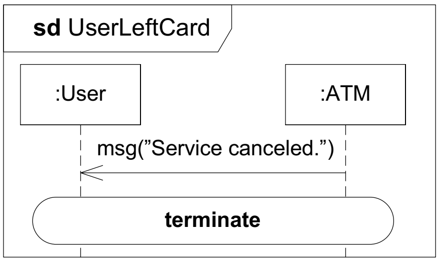


Fig. 10. Handling of left user

3.4 Time exceptions in Withdrawal

As shown we may attach possible exceptions to the constraint, in order to make it fire if the time constraint breaks. In Fig. 11 we apply this to a more complex example in order to highlight some challenging situations.

The exception that is of interest to us is the PIN validation timeout for the ATM. Notice that the AMTPinValidation exception uses three event semantics as described in TimedSTAIRS (see Section 2.3). This means that we accept that the message only needs to be received in the message buffer within the specified time constraint. We do not need to consume the message within the specified constraint. The reason for this time constraint is mainly to make sure that we don't lose contact with the bank during the request.

Fig. 12 specifies how an ATMPinValidationTimout exception is to be handled by the ATM and the Bank. The exception is triggered if the ATM doesn't receive the result of the PIN validation within the specified time. Our first exceptional reaction is to repeat the request to the Bank. If the response from the bank again fails to be delivered within the three time units, the ATMCancel exception specified in Fig. 12 is triggered.

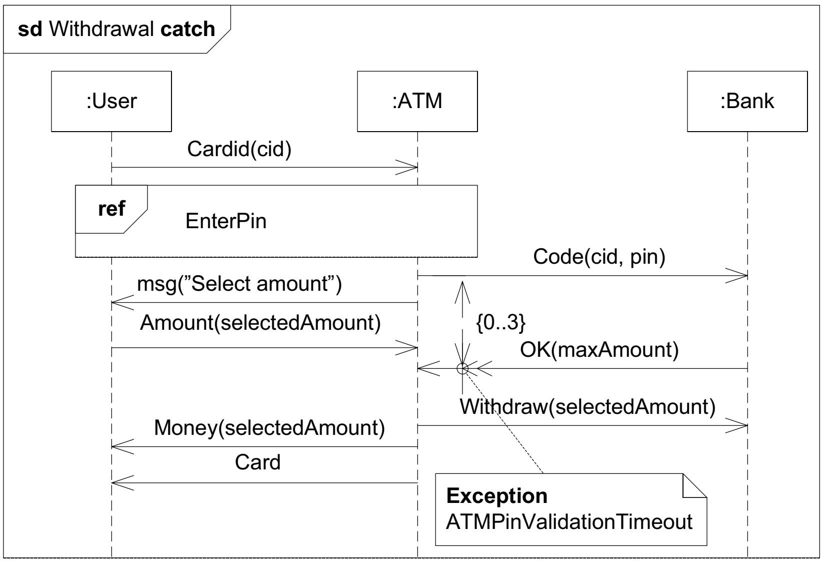


Fig. 11. Withdrawal with time exception

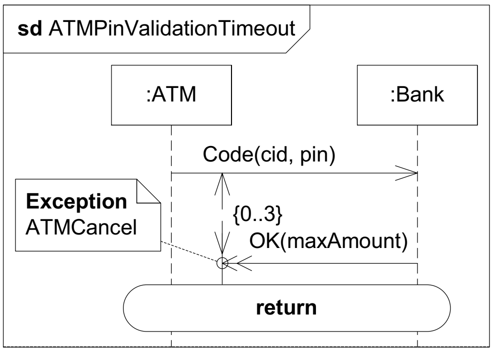


Fig. 12. Handling of PIN validation timeout on the ATM

Fig. 12 and Fig. 13 illustrate that an exception may end with return or with terminate. While returning means a perfect recovery back to the original flow of events, termination means that the service should be terminated gracefully. Termination concludes the closest invoker declaring catch as shown in Fig. 11. This means that if you declare an exception you must have a catch declaration as well.

We have so far said nothing about how exceptions behave in relation to the diagram that raised the exception. The basis for our exception handling notation is that we have some events that lead up to a possible exception. These are events

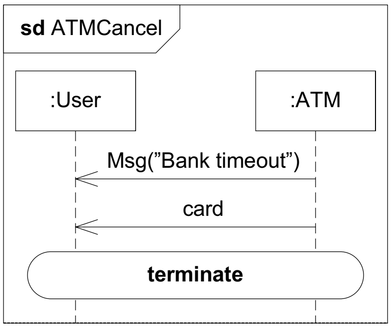


Fig. 13. The ATMCancel exception that terminates the Withdrawal

that must occur before a possible exception. Then we have events that can occur after a possible exception, and then we have the events enabled before a possible exception. These enabled events run in parallel with the exception handling.

If we apply this to Withdrawal, Fig. 11, we notice that the ATM must at least send a code for validation to the bank before the timeout event may occur. Actually the exception may only occur 3 time units after the sending of the validation request. That is, before the ATMPinValidationTimeout may occur the user must have given a card, entered the pin, the ATM must have sent the PIN for validation and 3 time units must have elapsed. After a possible recovery from the ATMPinValidationTimeout exception we can continue on with sending the withdrawal message and returning the card and money.

The tricky part is how to handle the selection of amount if an exception occurs. And as said, this will be done in parallel. That is because the user is outside of the ATMs sphere of control. We have in the ATM case 3 separate lifelines (User, ATM and Bank) that each communicate to the others through messages. Each lifeline in this distributed environment is totally autonomous meaning that one can not know which states the other lifelines are in. They are separate processes. We must therefore run the exception handling in parallel with other enabled events.

By enabled events we mean events that are triggered before the exception occurs. In the ATM example, an enabled event may be the sending of msg(“Select amount”), and all events triggered by that. These events are outside the control of the exception handling, and must be allowed to continue. An example of a non-enabled event is the sending of Money from the ATM. This event can never be sent before the OK message is received.

Compared to for example Java’s exceptions handling, Java’s exception handling is stack oriented. That means that exceptions may only occur inside a single thread of execution. Since sequence diagrams have several independent processes, we are unable to use the same approach.

We will continue to elaborate more on how exceptional traces are constructed in the following sections, keeping in mind that there are events that may occur in parallel with the exception handling.

4 The formal semantic domain of sequence diagrams

In this section we briefly recount the main parts of the semantics of timed sequence diagrams as defined in [3]. In the next section we give our proposal for how this semantics may be extended to handle time exceptions.

Formally, we use denotational trace semantics in order to capture the meaning of sequence diagrams. A trace is a sequence of events, representing one run of the system. As explained in Section 2.3, we have three kinds of events: the sending, reception and consumption of a message, denoted by !, $\sim$ and ? respectively. A message is a triple (s, tr, re) consisting of a signal s (the content of the message), a transmitter tr and a receiver re. The transmitter and receiver are lifelines, or possibly gates[3].

Each event in the sequence diagram has a unique timestamp tag to which real timestamps will be assigned. Time constraints are expressed as logical formulas with these timestamp tags as free variables. Formally, an event is a triple (k, m, t) of a kind k (sending, reception or consumption), a message m and a timestamp tag t.

As an example, Fig. 9 consists of six events: $(!, (m, ATM, User), t_1)$, $(\sim, (m, (ATM, User), t_2)$, $(?, (m, ATM, User), t_3)$, $(!, (d, User, ATM), t_4)$, $(\sim, (d, User, ATM), t_5)$ and $(?, (d, User, ATM), t_6)$ where m stands for the message msg("Enter PIN") and d stands for Digit. Notice that the reception events are implicit, meaning that they may happen at any time between the corresponding send and receive events. The given time constraint may now be written as $t_6 \leq t_1 + 5$.

$\mathcal{H}$ denotes the set of all well-formed traces. For a trace to be well-formed, it is required that

- for each message, the send event occurs before the receive event if both events are present in the trace.
- for each message, the receive event occurs before the consumption event if both events are present in the trace.
- the events in the trace are ordered by time.

$\mathcal{E}$ denotes the set of all syntactic events, and $[\![\ \mathcal{E}\ ]\!]$ is the set of all corresponding semantical events where real timestamps have been assigned to the timestamp tags:

$$[\![\ \mathcal{E}\ ]\!] \overset{\mathsf{def}}{=} \{(k, m, t \mapsto r) \mid (k, m, t) \in \mathcal{E} \wedge r \in \mathbb{R}\} \tag{1}$$

Informally, parallel composition $s_1 \parallel s_2$ of two trace-sets s_1 and s_2 is the set of all traces such that

[3] For a formal treatment of gates, see [4].

- all events from one trace in s_1 and one trace in s_2 are included (and no other events),
- the ordering of events from each of the traces is preserved.

Formally:

$$\begin{aligned} s_1 \parallel s_2 \overset{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \exists p \in \{1,2\}^\infty :& \\ \pi_2((\{1\} \times [\![\, \mathcal{E} \,]\!]) \textcircled{\scriptsize T} (p,h)) \in s_1 \wedge& \\ \pi_2((\{2\} \times [\![\, \mathcal{E} \,]\!]) \textcircled{\scriptsize T} (p,h)) \in s_2\}& \end{aligned} \tag{2}$$

The definition makes use of an oracle, the infinite sequence p, to resolve the non-determinism in the interleaving. It determines the order in which events from traces in s_1 and s_2 are sequenced. π_2 is a projection operator returning the second element of a pair, and $\textcircled{\scriptsize T}$ is an operator filtering pairs of sequences with respect to pairs of elements.

Weak sequencing, $s_1 \succsim s_2$, is the set of all traces obtained by selecting one trace h_1 from s_1 and one trace h_2 from s_2 such that on each lifeline, the events from h_1 are ordered before the events from h_2:

$$s_1 \succsim s_2 \overset{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in \mathcal{L} : h \upharpoonright l = h_1 \upharpoonright l \frown h_2 \upharpoonright l\} \tag{3}$$

where $\mathcal{L}$ is the set of all lifelines, $\frown$ is the concatenation operator on sequences, and $h \upharpoonright l$ is the trace h with all events not taking place on the lifeline l removed.

Time constraint on a trace set filters away all traces that are not in accordance with the constraint:

$$s \wr C \overset{\mathsf{def}}{=} \{h \in s \mid h \models C\} \tag{4}$$

where $h \models C$ holds if the timestamps assigned to timestamps tags in h does not violate C.

The semantics $[\![\, d \,]\!]$ of a sequence diagram d is given as a pair (p, n), where p is the set of positive and n the set of negative traces. Parallel composition, weak sequencing and time constraint are overloaded to such pairs as follows:

$$(p_1, n_1) \parallel (p_2, n_2) \overset{\mathsf{def}}{=} (p_1 \parallel p_2, (n_1 \parallel (p_2 \cup n_2)) \cup (n_2 \parallel p_1)) \tag{5}$$

$$(p_1, n_1) \succsim (p_2, n_2) \overset{\mathsf{def}}{=} (p_1 \succsim p_2, (n_1 \succsim (n_2 \cup p_2)) \cup (p_1 \succsim n_2)) \tag{6}$$

$$(p, n) \wr C \overset{\mathsf{def}}{=} (p \wr C, n \cup (p \wr \neg C)) \tag{7}$$

Two other operators, inner union ($\uplus$) and looping (μ_n) are also defined:

$$(p_1, n_1) \uplus (p_2, n_2) \overset{\mathsf{def}}{=} (p_1 \cup p_2, n_1 \cup n_2) \tag{8}$$

$$\mu_n\ (p, n) \overset{\mathsf{def}}{=} \begin{cases} (p, n) & \text{if n} = 1 \\ (\mu_{n-1}\ (p, n)) \succsim (p, n) & \text{otherwise} \end{cases} \tag{9}$$

Finally, the semantics of the sequence diagram operators of interest in this paper, is defined by:

$$\llbracket\, d_1 \ \mathsf{alt}\ d_2 \,\rrbracket \overset{\mathsf{def}}{=} \llbracket\, d_1 \,\rrbracket \uplus \llbracket\, d_2 \,\rrbracket \tag{10}$$

$$\llbracket\, d_1 \ \mathsf{par}\ d_2 \,\rrbracket \overset{\mathsf{def}}{=} \llbracket\, d_1 \,\rrbracket \parallel \llbracket\, d_2 \,\rrbracket \tag{11}$$

$$\llbracket\, d_1 \ \mathsf{seq}\ d_2 \,\rrbracket \overset{\mathsf{def}}{=} \llbracket\, d_1 \,\rrbracket \succsim \llbracket\, d_2 \,\rrbracket \tag{12}$$

$$\llbracket\, d_1 \ \mathsf{tc}\ C \,\rrbracket \overset{\mathsf{def}}{=} \llbracket\, d \,\rrbracket \wr C \tag{13}$$

$$\llbracket\, \mathsf{loop}\ (n)\ [d] \,\rrbracket \overset{\mathsf{def}}{=} \mu_n \llbracket\, d \,\rrbracket \tag{14}$$

Definitions of other operators may be found in e.g. [4].

5 The formal semantics of time exceptions

In Section 3 we informally explained the semantics of time exceptions. In this section we define the semantics formally, based on the formalism introduced in Section 4. Furthermore we show some desirable properties of our exception mechanisms.

5.1 Definitions

An exception diagram is mainly specified using the same operators as ordinary sequence diagrams, and its semantics may be calculated using the definitions given in Section 4. As explained in Section 3, the additional constructs used in exception diagrams is that the exception handling always ends with either return or terminate. Formally, the semantics of an exception (sub-)diagram marked with either return or terminate is defined by:

$$\llbracket\, d \ \mathsf{return} \,\rrbracket \overset{\mathsf{def}}{=} \llbracket\, d \,\rrbracket \tag{15}$$

$$\llbracket\, d \ \mathsf{terminate} \,\rrbracket \overset{\mathsf{def}}{=} \mathit{appendTT}(\llbracket\, d \,\rrbracket) \tag{16}$$

where $\mathit{appendTT}$ is a function appending a special termination event TT to every trace in its operand (i.e. all the positive and negative traces described by d).

With this new termination event, weak sequencing of trace sets must be redefined so that traces that end with termination are not continued:

$$\begin{aligned} s_1 \succsim s_2 \overset{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 :& \\ (\mathit{term}(h_1) \wedge h = h_1) \ \vee& \\ (\neg \mathit{term}(h_1) \wedge \forall l \in \mathcal{L} : h \restriction l = h_1 \restriction l \frown h_2 \restriction l)\}& \end{aligned} \tag{17}$$

where $\mathit{term}(h_1)$ is a boolean function that returns true if h_1 ends with the termination event TT, and false otherwise.

For parallel composition of trace sets, the traces may be calculated as before and then removing all events that occur after TT from the trace:

$$s_1 \parallel s_2 \overset{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \exists h' \in s_1 \parallel' s_2 : h = \mathit{chopTT}(h')\} \tag{18}$$

where $\parallel'$ is parallel composition as defined by definition 2 and chopTT is a function removing all events after TT in the trace (if it exists).

A sequence diagram d marked as catching termination events then has the semantic effect that the termination mark is removed from the trace, meaning that the trace continues as specified by the diagram that is enclosing d:

$$[\![\, d \; \mathsf{catch} \,]\!] \overset{\mathsf{def}}{=} \mathit{removeTT}([\![\, d \,]\!]) \tag{19}$$

where $\mathit{removeTT}$ is a function removing the special termination event TT from all traces in its operand.

Finally, we need to define the semantics of a sequence diagram which contains exceptions. The kind of exceptions considered in this paper is always connected to an event with a time constraint. Syntactically, we write q tc C $\mathsf{exception}$ e to specify that C is a time constraint associated with the event q, and that the sequence diagram e specifies the exception handling in case C is violated. Obviously, a trace is negative if the exception handling starts before the time constraint is actually violated.

As an example, consider the sequence diagram in Fig. 9. Here, we have the constraint $t_6 \leq t_1 + 5$ as explained in Section 4. Letting t_7 be the timestamp of the sending of the message in Fig. 10, we then intuitively have the corresponding constraint $t_7 > t_1 + 5$.

Formally, we let $\mathit{frst}(e)$ be the set of all events that may be the first event in a trace in the semantics of e. (These events should all be send-events on the same lifeline as q, as this is the lifeline that discovers the time constraint violation and then initiates the exception handling.) We then define $e_{q,C}$ as the exception diagram e where every event q' in $\mathit{frst}(e)$ is replaced by q' tc $\neg C[\mathit{tt}(q')/\mathit{tt}(q)]$ where tt is a function returning the timestamp tag of an event and $C[x/y]$ is the constraint C with x substituted for y. In other words, $e_{q,C}$ is the exception diagram where the time constraint C on q has been transformed into corresponding time constraints for the first events in the exception handling.

The semantics of a sequence diagram with an exception is then defined by:

$$\begin{aligned} &[\![\, d_1 \; \mathsf{seq} \; (q \; \mathsf{tc} \; C \; \mathsf{exception} \; e) \; \mathsf{seq} \; d_2 \,]\!] \overset{\mathsf{def}}{=} \\ &\quad [\![\, d_1 \; \mathsf{seq} \; (q \; \mathsf{tc} \; C) \; \mathsf{seq} \; d_2 \,]\!] \uplus \\ &[\![\, (d_1 \; \mathsf{seq} \; d_2) \; \mathsf{par} \; e_{q,C} \,]\!] \; ⓢ \, \{h \in \mathcal{H} \mid h \restriction \mathit{ll}(q) \in [\![\, d_1 \; \mathsf{seq} \; e_{q,C} \; \mathsf{seq} \; d_2 \,]\!] \restriction \mathit{ll}(q)\} \end{aligned} \tag{20}$$

where ll is a function returning the lifeline of an event (the sender in the case of a send event, the receiver in the case of a reception or a consumption event), ⓢ is a filtering operator such that $(p, n) ⓢ S$ is the pair (p, n) where all traces that are not in the set S are removed, $h \in (p, n)$ is a short-hand for $h \in p \vee h \in n$, and $\restriction$ is overloaded from traces to pairs of sets of traces in standard pointwise manner.

In definition 20, the first part corresponds to the semantics without the exception. In essence, this gives as positive the traces of $[\![\ d_1 \ \mathsf{seq}\ q\ \mathsf{seq}\ d_2\ ]\!]$ where q has a timestamp that is valid according to C, and as negative the traces where the timestamp of q is invalid. The second part is all traces where q has not occurred, and the exception handling is performed instead. The use of par means that the exception handling may be performed in parallel with the system trying to continue its ordinary behavior. This is natural for all other lifelines than the one where the exception occurred, as they are not aware of the exception. All events that may happen without involving the lifeline with the exception, may still happen after the exception is triggered.

The lifeline where the exception occurred is the lifeline of q, written $ll(q)$. For this lifeline, it is necessary that the exception handling should be performed before it continues with the ordinary behavior. Hence, the effect of the filtering operator is to remove all traces where this is not the case.

Looking again at the sequence diagram in Fig. 9, one possible scenario would be that the ATM sends the message msg("Enter Pin"), but the user for some reason never receives this messages. After having waited five time units, the ATM then sends the message msg("Service cancelled"), the user receives and consumes the message, and the trace terminates. This corresponds to the trace $\langle !msg(EP), !msg(SC), \sim msg(SC), ?msg(SC)\rangle$. If the timestamp of $!msg(SC)$ is more than five time units greater that the timestamp of $!msg(EP)$ the trace is positive, otherwise the trace is negative. However, a scenario where the ATM sends the message msg("Service Cancelled") before it sends msg("Enter Pin") is not described by the diagram, and all such traces would be inconclusive.

Theorem 1. *Assuming that no event occurs more than once in the sequence diagrams, we have associativity with respect to exceptions, i.e.*

$$\begin{aligned}&[\![\ d_1\ \mathsf{seq}\ (q_1\ \mathsf{tc}\ C_1\ \mathsf{exception}\ e_1)\ \mathsf{seq}\ (d_2\ \mathsf{seq}\ (q_2\ \mathsf{tc}\ C_2\ \mathsf{exception}\ e_2)\ \mathsf{seq}\ d_3)\ ]\!]\\ =\ &[\![\ (d_1\ \mathsf{seq}\ (q_1\ \mathsf{tc}\ C_1\ \mathsf{exception}\ e_1)\ \mathsf{seq}\ d_2)\ \mathsf{seq}\ (q_2\ \mathsf{tc}\ C_2\ \mathsf{exception}\ e_2)\ \mathsf{seq}\ d_3\ ]\!]\end{aligned}$$

This is proved in [2].

□

5.2 Refinement

TimedSTAIRS [3] defines supplementing and narrowing as two special cases of refinement. Supplementing means to add more traces as positive or negative to the sequence diagram, while narrowing means to redefine earlier inconclusive traces as negative. Formally, a sequence diagram d' with semantics (p', n') is said to be a refinement of another sequence diagram d with semantics (p, n), written $d \rightsquigarrow d'$, iff

$$n \subseteq n' \ \wedge\ p \subseteq p' \cup n' \tag{21}$$

It should be clear from our explanations in Section 3 that adding exception handling to a sequence diagram constitutes a refinement. Adding a time constraint is an example of narrowing, as traces with invalid timestamps are moved from positive to negative when introducing the time constraint. More generally, we have the following theorem:

Theorem 2. *Assuming that the exception diagram e is not equivalent to the triggering event q, i.e. $[\![\ e\ ]\!] \neq (\{\langle q \rangle\}, \emptyset)$, we have that*

1. d_1 seq q seq $d_2 \rightsquigarrow d_1$ seq $(q$ tc $C)$ seq d_2
2. d_1 seq $(q$ tc $C)$ seq $d_2 \rightsquigarrow d_1$ seq $(q$ tc C $\mathsf{exception}$ $e)$ seq d_2
3. d_1 seq q seq $d_2 \rightsquigarrow d_1$ seq $(q$ tc C $\mathsf{exception}$ $e)$ seq d_2

Proof.

1. Straightforward from definition 13 of time constraint, which ensures that all traces of the original diagram are also traces of the diagram with the time constraint added.
2. Straightforward from definition 20 of an exception, as the semantics of the original diagram is included as the first part of the definition.
3. Follows directly from the two previous facts using that refinement is transitive (proved in [4]).

□

Finally, the following theorem demonstrates that for a diagram containing exceptions, the normal and exceptional behavior may be refined separately:

Theorem 3. *Refinement is monotonic with respect to exceptions as defined by definition 20, i.e.:*

$$d_1 \rightsquigarrow d_1' \wedge d_2 \rightsquigarrow d_2' \wedge e \rightsquigarrow e' \Rightarrow$$

$$d_1 \ \mathsf{seq}\ (q\ \mathsf{tc}\ C\ \mathsf{exception}\ e)\ \mathsf{seq}\ d_2 \rightsquigarrow d_1'\ \mathsf{seq}\ (q\ \mathsf{tc}\ C\ \mathsf{exception}\ e')\ \mathsf{seq}\ d_2'$$

This is proved in [2].

□

6 Conclusions

We have shown that introducing time exceptions improve the completeness of sequence diagram descriptions while keeping the readability of the main specification. We have defined concrete notation for exceptions built on existing symbols of UML 2 and the simple time notation. Finally, we have given a precise formal definition of time exceptions and shown that our concepts are compositional since refinement is monotonic with respect to exceptions.

References

1. Oddleif Halvorsen and Øystein Haugen. Proposed notation for exception handling in UML 2 sequence diagrams. In *Australian Software Engineering Conference (ASWEC)*, pages 29–40. IEEE Computer Society, 2006.
2. Oddleif Halvorsen, Ragnhild Kobro Runde, and Øystein Haugen. Time exceptions in sequence diagrams. Technical Report 344, Department of Informatics, University of Oslo, September 2006.
3. Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 1–25. Springer, 2005.
4. Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. Technical Report 309, Department of Informatics, University of Oslo, 2005.
5. Object Management Group. *OMG Unified Modeling Language 1.4*, 2000.
6. Object Management Group. *UML profile for Schedulability, Performance and Time Specification*, document: ptc/05-01-02 edition, 2005.
7. Object Management Group. *UML Testing Profile*, document: ptc/05-07-07 edition, 2005.
8. Object Management Group. *UML 2.1 Superstructure Specification*, document: ptc/06-04-02 edition, 2006.

An Approach to Performance Modeling of Software Product Lines

Julie A. Street[1]and Hassan Gomaa[2]

[1] The Aerospace Corporation,
Chantilly, Virginia 22151 (USA)
julie.a.street@aero.org
[2] George Mason University
Fairfax, Virginia 22030 (USA)
hgomaa@gmu.edu

Abstract: It is often the case that software performance requirements are not addressed until after development and sometimes even after deployment. This usually leads to late changes to the software, which can impact schedule, cost, and quality, or necessitate the acquisition of high performance more expensive hardware. However, if the performance requirements are properly captured and analyzed during design, areas where the system does not meet its performance requirements can be identified and remedied early, which results in less costly changes. Performance is especially important in software product lines because one design is used for many software applications. The goal of this paper is to develop an approach for capturing software performance characteristics in software product line (SPL) requirements, analysis, and design models. The approach extends the Object-Oriented Analysis and Design for Product Lines (PLUS) method to use the UML Profile for Schedulability, Performance and Time (SPT). This paper discusses the necessary extensions, which include changes to use case descriptions, feature models, interaction diagrams, and deployment diagrams. These changes are illustrated using a cruise control and monitoring software product line case study.

1 Introduction

Software performance is a complex issue that depends on many aspects of a software system. Some performance measures for a system include, “resource utilization, waiting times, execution demands (for CPU cycles or seconds), and response time, (the actual or wall-clock time to execute a scenario step or scenario)” [1]. It is often the case that performance requirements are not addressed until after the application is developed and sometimes even after it is deployed. This usually results in changes to the software or the acquisition of high performance hardware, both of which can impact cost, schedule, and quality. For example, National Aeronautics and Space

Association's (NASA) found performance issues in its Flight Operations Segment (FOS), which was designed for many functions including commanding, planning, and scheduling. It was reported that the FOS failed performance requirements on response times for developing satellite schedules and analyzing satellite status and telemetry data, which led to a costly eight-month delay in launch of the program [2]. Another real world example is from the London Ambulance Service Automated Vehicle Locating System (AVLS) deployed in 1993. During deployment, the software performance degraded because of a large number of exception messages and repeat calls from customers [3]. The two examples discussed demonstrate the costly real world impacts of not addressing software performance requirements until after development and deployment.

To address the problems discussed in the examples, researchers developed performance modeling and analysis techniques for addressing performance requirements early in the software lifecycle. Performance modeling and analysis techniques model the software before development and predict the software's performance. The predicted performance data is analyzed to determine if the software will meet the performance requirements. If it does not meet the performance requirements, then changes are made to the design, architecture, or performance requirements. Changes made early in the software lifecycle are typically less costly because they do not require code or infrastructure changes. After the changes are made, the design is reanalyzed. The process of changing and reanalyzing is repeated until the performance requirements are met. Once the performance requirements are met, development can begin on the optimized design.

Addressing performance requirements is even more critical in software product lines because one software product line (SPL) design will result in multiple individual systems called SPL members. A software product line (SPL) is a family of software systems that share a set of common features, and each system can have additional functionality. The purpose of designing a SPL is to reuse software and design models, which ultimately saves time and effort in building the individual systems.

When compared to single systems, SPLs have additional complexities with respect to functional and performance requirements. First, functionality and performance requirements may vary from SPL member to SPL member; therefore, this variability must also be taken into account. Secondly, the amount of performance analysis increases because of all the potential SPL member and deployment environment combinations that can result from a single SPL design. This is because performance modeling is platform dependent; therefore, each configuration is analyzed separately. In the case of SPL, this can quickly lead to a combinatorial explosion and significantly increase the amount of performance analysis required. To avoid combinatorial explosion, analysis can be conducted on a SPL member-by-SPL member bases or on common configurations. While conducting quantitative analysis to optimize a SPL is a crucial step, it is not the focus of this paper. The focus of this paper is to outline how to capture performance requirements in design in order to conduct quantitative analysis on the design. This paper proposes an approach to effectively capture performance requirements and data in SPL requirements, analysis, and design models. This paper discusses the necessary extensions, which include changes to use case descriptions, feature models, interaction diagrams, and deployment diagrams.

This paper's contribution is to fill the gap between software performance modeling and software product lines. The structure of the paper is as follows; Section 2 discusses related work to this paper. Section 3 discusses the extensions to PLUS for modeling performance requirements in SPL requirements, analysis, and design models. Section 4 discusses the main conclusions of the paper.

2 Related Work

Many approaches have been developed to model software performance directly in software designs. Extensions have been created for the Unified Modeling Language (UML), a standard modeling language for object oriented applications, to address performance. This is because UML by itself lacks the concepts to properly capture performance requirements. Some performance-related extensions include UML Profile for Schedulability, Performance and Time (SPT) [1], UML-Real Time (RT) [4], Hard Read Time (HRT)-UML [5], Embedded UML [6], ACCORD/UML [7] and more. However none of these approaches address the complexities and variability associated with software product lines.

Performance techniques outside of UML have also been developed. The Hard Real-Time Hierarchal Object Oriented Design (HRT HOOD) was developed by the European Space Agency for designing hard real-time applications. It uses five predefined object types with various privileges and constraints on the objects. This makes it possible to conduct timing analysis on the system [8]. Wu et al have proposed a reusable component-based approach in which components with known performance characteristics can be reused. This way the software performance can be more easily predicted [9], [10]. While both these approaches have their advantages and promote reuse, they do not directly address software product lines and software product line complexities.

3 Performance Modeling Method for SPLs

The approach taken in this paper is to extend the Object-Oriented Analysis and Design method for Product Lines (PLUS) [11] with the Performance subprofile of the UML Profile for Schedulability, Performance and Time (SPT). The novelty of this approach is the ability to support performance modeling, while maintaining the variability necessary for creating SPL designs. PLUS [11] is a method used for designing software product lines that utilizes Unified Modeling Language (UML) 2.0. It is divided into the four phases; Requirements Modeling, Analysis Design, Architectural Design, and Application Engineering. This method is extended with the SPT profile to enable the modeling and analysis of performance requirements. The

SPT profile is an extension of UML 1.4 that defines a standard paradigm for modeling time, schedulability, and performance-related aspects of real-time systems [1]. In the proposed method, UML 2.0 terminology is used, even though the UML SPT Profile is an extension of UML 1.4. This is because UML 2.0 is the latest standard, and any future extensions of this work will utilize UML 2.0. The following subsections describe in detail the major extensions to PLUS in the requirements and design modeling phases of SPL development, and in application engineering, when an application is derived from the SPL models.

3.1 PLUS with SPT Requirements Modeling Extensions

The first phase of the PLUS with SPT method is requirements modeling. The goal of requirements modeling is to capture the functional and performance requirements of the system. In order to effectively capture software performance requirements in SPL requirement models extensions were needed to the use case descriptions, and feature models. The subsequent paragraphs describe the details of these changes.

The first extension is to the use case descriptions. Use cases diagrams and descriptions are used to capture the SPL's common and variable functionality. SPL requirements are different from single system requirements because not all the requirements will be implemented in all SPL members [11]. When conducting software performance modeling, software performance requirements, which are requirements that focus on system performance measures such as resource utilization and response times, also need to be documented in the use case descriptions. It is reasonable to assume that not all SPL members are subject to the same performance requirements. Therefore, performance requirements are also subject to variability. To capture software performance requirements and their variability, a performance requirement section is added to the use case descriptions. This section contains the performance requirement name, description, variability, and reference to the use case it applies to.

To see illustrate the use case description extension, consider a cruise control and monitoring SPL system that will be used on multiple cars as shown in Figure 1. Common functionality required by all SPL members is denoted with PLUS's <<kernel>> reuse category, and variable functionality is marked with the <<optional>> or <<alternative>> reuse category [11]. Use case descriptions are created to clarify the use case diagram and are extended to document performance requirements. In the cruise control and monitoring SPL, there is a performance requirement for the system's response time to the driver actions. This is captured in the performance requirement section of use case description, as seen in Figure 2 lines 6-11. This particular performance requirement has variability because high performance cars need to respond faster than standard cars. This is modeled as a performance requirement variation point called 'High Performance Response Time' as shown in Figure 2 lines 12-17. It is marked with the optional alternative reuse category, meaning that either the standard or high performance response time will be used, but not both.

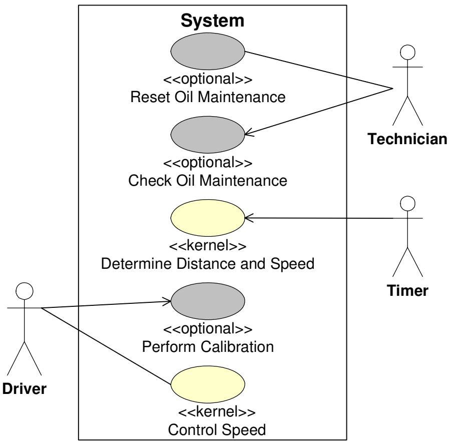


Fig. 1. Cruise Control and Monitoring System Use Case Model

1 **Use Case Name:** Control Speed
2 **Reuse Category:** kernel
3 **Description:** This use case is described in terms of a typical scenario consisting
4 of the following sequence of external events.
5 1. During any event change initialed by driver actions, …
6 **Performance Requirements**
7 **Name:** Standard Performance Driver Response Time
8 **Type of requirement**: default
9 **Line number(s):** 5
10 **Description**: The system must respond to the driver's actions within 500msec
11 for standard cars.
12 **Performance Requirement Variation Points**
13 **Name:** High Performance Driver Response Time
14 **Type of requirement:** optional alternative
15 **Line number(s):** 5
16 **Description of functionality:** On high performance cars, the system must
17 respond to the driver's actions within 250msec.

Fig. 2. Section of Control Speed Use Case Description

The other extension to the requirements modeling phase to support software performance modeling is to the feature model. A feature is a characteristic of the software that one or more of the SPL members will provide. The feature model consists of a feature-use case traceability table and a feature dependency diagram [11]. The feature-use case traceability table captures features and their characteristics in a tabular format. A feature dependency diagram is created using a class diagram. Features are modeled as metaclasses, and associations are added to identify the relationships between the various features. Associations are used instead of dependencies because they offer greater modeling capabilities [11]. In order to support software performance modeling, performance features, which are timing characteristics of the system, also need to be considered in the feature model. This will help designers identify the relationships between features and performance features to understand the performance impacts of including different functionality.

In PLUS [11], functional features for a product line can be <<common>>, <<optional>>, <<alternative>>, or <<default>>. In addition, a feature group represents a group of features with a particular constraint on their usage in a SPL member, e.g., to represent mutually exclusive features. To distinguish performance features from functional features, additional feature group and feature stereotypes are needed. Additionally, performance features are subject to variability, therefore the stereotypes also must reflect this variability. To help distinguish performance features from functional features and capture their variability, the following stereotypes are added; <<common performance feature>>, <<optional performance feature>>, <<alternative performance feature>>, <<default performance feature>> and <<parameterized performance feature>>. These stereotypes are an extension of the UML metaclass and stereotype notation for features used in the PLUS method.

To illustrate the extensions to feature modeling, consider the cruise control and monitoring SPL. The features that are part of all SPL members are captured in one common feature called 'Cruise Control System Kernel,' as shown in Table 1 row one.

Table 1. Selected Portion of Feature/Use Case Traceability Table

No.	Feature Name	Feature Category	Use Case Name	Use Case Category/ VP	VP Name
1	Cruise Control System Kernel	common	Control Speed	kernel	
2	Calculate Calibration	optional	Perform Calibration	optional	
3	Record Start/Stop Response Time	optional performance	Perform Calibration	optional	
4	Compute Calibration Execution Time	optional performance	Perform Calibration	optional	
5	Standard Response Time	default performance	Control Speed	vp	Cruise Control Response Time
6	High Performance Response Time	alternative performance	Control Speed	vp	Cruise Control Response Time

The use case 'Perform Calibration' has two performances needs: response time and execution time. Therefore two performance features, 'Compute Calibration Execution Time' and 'Record Start/Stop Response Time,' are added to the table. 'Perform Calibration' is an optional use case; therefore, the performance features are also optional, and they are marked with the new <<optional performance feature>> stereotype as shown in the 'Feature Category' column of Table 1 rows three and four. The 'Control Speed' use case contains two alternative performance requirements for the system's response time to a driver's actions. This type of variation can be captured with the new <<default performance feature>> and <<alternative performance feature>>, as shown in the 'Feature Category' column of Table 1 rows five and six.

After all the features are identified in the traceability table, the feature dependency diagram is created. In the case study, the 'Calculate Calibration' optional feature, if implemented must meet the two optional performance requirements. Therefore, there is a mutually inclusive relationship between 'Calculate Calibration' and 'Record Start/Stop Response Time' and 'Compute Calibration Execution Time', as shown in Figure 3. This dependency is not obvious by just looking at the traceability table, which is the motivation for building the feature dependency model.

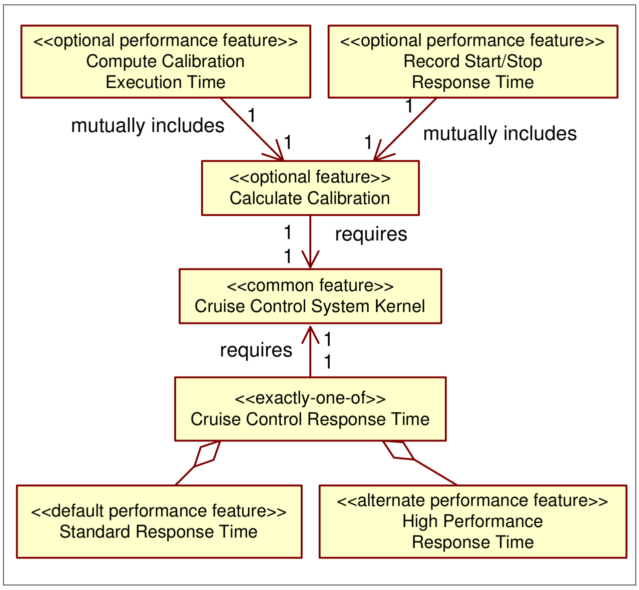

<<optional performance feature>> Compute Calibration Execution Time
<<optional performance feature>> Record Start/Stop Response Time
mutually includes
1
1
1
1
mutually includes
<<optional feature>> Calculate Calibration
1
1
requires
<<common feature>> Cruise Control System Kernel
requires
1
1
<<exactly-one-of>> Cruise Control Response Time
<<default performance feature>> Standard Response Time
<<alternate performance feature>> High Performance Response Time

Fig. 3. Examples of Feature Dependency Modeling

The 'Cruise Control Response Time' feature group, seen in the 'VP column' of Table 1 rows five and six, contains two alternative performance features. The feature group is modeled with the composition relationship, as shown in Figure 3. In this feature group, only one of the features can be selected to implement, therefore it is given the <<exactly-one-of>> PLUS stereotype.

3.2 PLUS with SPT Design Modeling Extensions

The architectural design phase of the PLUS with SPT method consists of defining the software architecture. This phase involves creating the concurrent communication diagrams, subsystem structure, and establishing the component interfaces [11]. To support software performance modeling, the concurrent communication diagrams and deployment diagrams are extended to support capturing of platform independent performance data and in some cases platform specific performance data. The subsequent paragraphs describe the se extensions in detail.

First, concurrent communication diagrams, which place an emphasis on concurrent behavior, are extended to include platform independent performance data. Concurrent communication diagrams are different from standard communication diagrams because they must explicitly show the concurrent behavior using UML 2.0 active objects, passive objects, synchronous messages and asynchronous messages [11]. Then platform independent performance data is added using the UML SPT performance subprofile stereotypes.

Platform independent performance data is added by first marking all concurrent communication diagrams that will have performance characteristics with the performance context stereotype of <<PAcontext>>. All diagrams must maintain their reuse categories to maintain the SPL variability. Some concurrent communication diagrams may cover multiple scenarios. In these situations, the scenarios must be separated in to individual diagrams based on SPT constraints. Next, the steps (e.g. message exchanges, processing actions, etc) in the scenario are tagged with step stereotypes corresponding to their behavior. The first step defines the scenario and outlines how requests for the scenario will occur. This first step is always marked with either a <<PAopenLoad>> or <<PAclosedLoad>> stereotype. The <<PAopenLoad>> stereotype is used when there is a stream of new inputs at a known rate, and the <<PAclosedLoad>> is used for when there are fixed number of inputs that cycle through the system with a delay between each cycle. In addition to the stereotype, the first step should also contain the requirement data in the stereotypes attribute tags. It should be captured in the <attribute-tag>::= <source-modifier><type-modifier><time-value> format. The source-modifier must be set to ‘req’ to denote a requirement, the type modifier contains the statistical information (if any), and the time value contains the value and units [1]. After the first step, all subsequent steps are annotated with the <<PAstep>> stereotype. Once all the steps are tagged, the platform independent performance data is complete.

To illustrate adding platform independent data, consider the concurrent communication diagram for the system’s response to the driver’s actions from the cruise control and monitoring SPL. This scenario has two performance requirements on the systems response; therefore it is marked with the <<PAcontext>> to identify it as a performance scenario. All SPL members require this scenario, therefore it is also marked with the <<kernel>> reuse category, as shown in Figure 4.

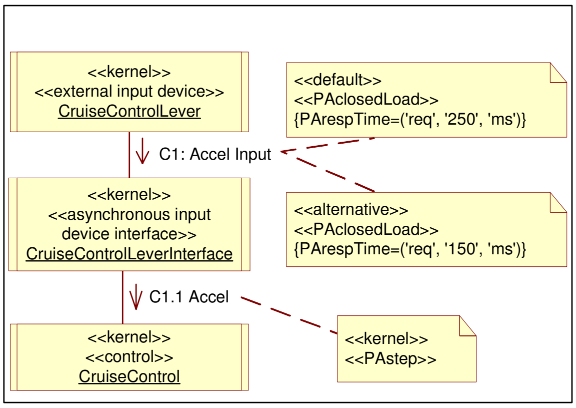


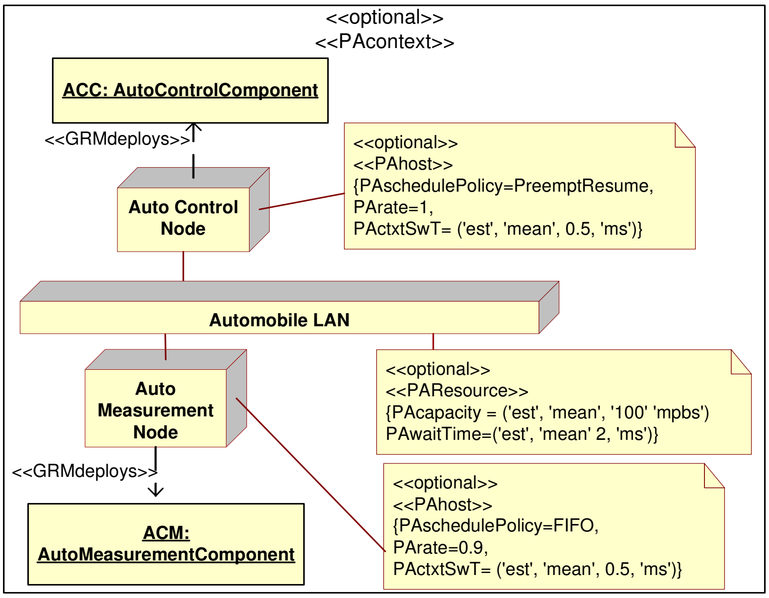
Fig. 4. Portion of Concurrent Communication Diagram with Platform Independent Data

The first step in this scenario is the driver using the cruise control lever. This input is modeled with the <<PAclosedLoad>> stereotype because there is only one potential input that comes from the lever and a delay between each input. It has two potential performance requirements associated with it. Therefore two sets of stereotypes are added to show the default and alternative requirements as shown in Figure 4. The performance requirements are on the response time; therefore the PArespTime tag is used with the source-modifier set to 'req'. All subsequent steps are modeled using the <<PAstep>> stereotype and <<kernel>> reuse category since all systems must perform this scenario. After all steps in the scenario have performance and reuse stereotypes, the platform independent modeling is complete.

The second major extension to the architectural modeling phase is to capture platform specific performance information for common or known SPL member configurations. Each configuration requires a deployment diagram and a set of concurrent communication diagrams because software performance can vary significantly on different configurations. The deployment diagram describes the configuration, shows a mapping of objects to hardware, and captures hardware performance information. The concurrent communication diagram captures software performance data in the context of a scenario. The subsequent paragraphs describe the process for create these platform specific diagrams.

Creating platform specific deployment diagrams involves tagging the deployment diagram with UML SPT performance subprofile stereotypes. First, any deployment diagram that contains performance data should be marked with the <<PAcontext>> stereotype and appropriate reuse category. Next, all the resources are tagged with the appropriate reuse category and SPT stereotypes and attributes. Processing resources, which are devices or interface devices with processing steps associated with them, are tagged with the <<PAhost>> stereotype. Then, platform specific performance data, such as throughput rate, and scheduling policy are added in the <<PAhost>>'s attribute tags. Passive resources, which are protected devices or entities that may be shared by concurrent operations, are denoted with the <<PAresource>> stereotype. Then performance data such as scheduling policy, capacity, waiting time, and more are captured in the attribute tags. After all the system resources are marked with the appropriate stereotype and performance data, the mapping between hardware and software is defined. This can be captured through the <<GRMdeploys>> association between a node and a component instance on the deployment diagram [1]. After all the mappings between hardware and software are modeled, the deployment diagram with platform specific performance data is complete.

To illustrate the process of creating deployment diagram with platform specific performance data consider the cruise control and monitoring SPL system. Figure 5 shows a portion deployment diagram with platform specific performance data, which is identified by the <<PAcontext>> stereotype. Since all SPL members will not use this deployment, it is also marked with the <<optional>> reuse category.



<<optional>>
<<PAcontext>>
ACC: AutoControlComponent
<<GRMdeploys>>
Auto Control Node
<<optional>>
<<PAhost>>
{PAschedulePolicy=PreemptResume,
PArate=1,
PActxtSwT= ('est', 'mean', 0.5, 'ms')}
Automobile LAN
Auto Measurement Node
<<optional>>
<<PAResource>>
{PAcapacity = ('est', 'mean', '100' 'mpbs')
PAwaitTime=('est', 'mean' 2, 'ms')}
<<GRMdeploys>>
<<optional>>
<<PAhost>>
{PAschedulePolicy=FIFO,
PArate=0.9,
PActxtSwT= ('est', 'mean', 0.5, 'ms')}
ACM: AutoMeasurementComponent

Fig. 5. Portion of a Deployment Diagram with Platform Specific Performance Data

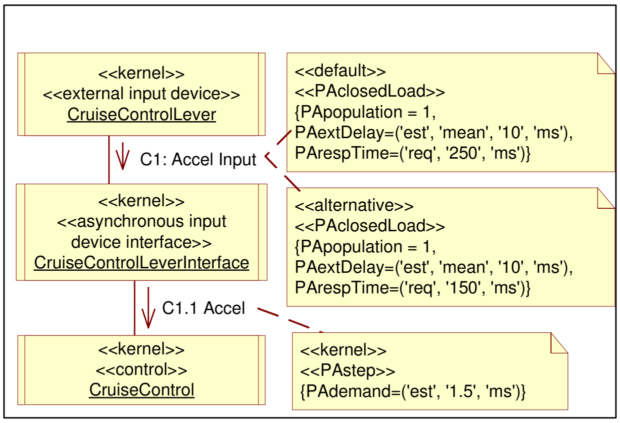
This diagram has two nodes that are processing resources, and it will not be used in all SPL members. Therefore they are marked with the <<PAhost>> and <<optional>> stereotypes. The performance characteristic for each of these nodes is captured using the stereotype's tags. For example, the Auto Measurement node uses the PAschedulePolicy='FIFO' tag to show this node processes in first in first out fashion. The PArate represents relative speed expressed as a percentage of some standard processor, and is 0.9 in this deployment. Finally, the context switching time is captured for this node using the PActxtTime tag. The Automobile LAN is a resource that is shared among concurrent operations, and it will not be used in all SPL members. Therefore, it is marked with the <<PAresource>> and <<optional>> stereotypes. Its performance characteristics are wait times and capacity, which are captured in PAwaitTimes and PAcapacity respectively.

Next, the mappings between hardware and software are defined. The Auto Control Component will be deployed on the Auto Control Node, therefore an association is drawn between them and marked with the <<GRMdeploys>> stereotype as shown in Figure 5. After all the mappings are modeled the platform specific performance modeling on the deployment diagrams is compete.

The second part of creating the platform specific performance models is creating a set of platform specific performance concurrent communication diagrams. The process for adding platform specific performance data is to create a new set of diagrams from the platform independent concurrent communication diagrams. Then, platform specific performance data is added to the SPT stereotype's attribute tags. Developing estimates for the software performance of each potential platform configuration is not a trivial task. Techniques to help in the estimation process can be found in [12] and [13]. All of the attribute tags do not need to be used, only the attribute tags that will be used in the corresponding quantitative analysis are needed. The platform specific data should be captured in attribute tags data type. A frequently used format is <attribute-tag>::=<source-modifier><type-modifier><time-value>. The source-modifier contains how the value was determined, such as predicted or measured. The type-modifier is used to define the statistical meaning, for example percentile. Finally, the time-value is the actual value [1].

To illustrate modeling platform specific performance data, consider the concurrent communication diagram with platform independent data from Figure 4, running on the deployment configuration in Figure 5. This scenario will be analyzed using event sequence analysis to determine the performance requirement is met. Therefore the required platform specific performance data is the closed workload's PApopulation for the number of users and PAextDelay for the delay between inputs. Additionally, each step will need the PAdemand for the CPU time, PAdelay for any delay time, and PAexOp for any external operation time. In this scenario, the closed workload is the cruise control lever. There is only one lever on a car; therefore PApopulation is set to one on the <<PAclosedLoad>> stereotype. The frequency with which the lever is pushed is estimated and captured in the PAextDelay, as shown in Figure 6. The source-modifier is set to 'est' since the rate is an estimated value.

Next, performance data is added to each of the steps. The required time for each subsequent step is estimated and captured in the PAdemand, PAdelay, and PAexOp attribute tags in the <attribute-tag>::=<source-modifier><type-modifier><time-value> format. The step shown in Figure 6 does not have any associated delays or external operations; therefore the attribute tags are omitted. Once all the steps have their associated performance data, the platform specific modeling is complete.



<<kernel>>
<<external input device>>
CruiseControlLever
C1: Accel Input
<<kernel>>
<<asynchronous input device interface>>
CruiseControlLeverInterface
C1.1 Accel
<<kernel>>
<<control>>
CruiseControl
<<default>>
<<PAclosedLoad>>
{PApopulation = 1,
PAextDelay=('est', 'mean', '10', 'ms'),
PArespTime=('req', '250', 'ms')}
<<alternative>>
<<PAclosedLoad>>
{PApopulation = 1,
PAextDelay=('est', 'mean', '10', 'ms'),
PArespTime=('req', '150', 'ms')}
<<kernel>>
<<PAstep>>
{PAdemand=('est', '1.5', 'ms')}

Fig. 6. Portion of Concurrent Communication Diagram with Platform Specific Data

3.3 PLUS with SPT Application Engineering

The final phase in PLUS with SPT method is the application engineering, which is the process for determining a SPL member. This phase utilizes the SPL artifacts to derive the individual SPL member. If a SPL member is using a common or known configuration that already has the platform specific diagrams captured in the SPL design, those diagrams can be reused and tailored to the specific SPL member. An advantage to using a common configuration is that as the SPL members using the common configuration are developed, the diagrams can be updated with actual performance metrics to help improve the accuracy of future quantitative analysis. If a SPL member is not using a common configuration, then new platform specific diagrams must be created.

After the platform specific diagrams are completed, quantitative analysis can be conducted on the model to determine if the performance requirements can be met. To illustrate how the data can be pulled from the platform specific models and used in quantitative analysis, consider the case study in Figure 5 and Figure 6 using the default performance requirement.

In the example discussed above, event sequence analysis will be used to determine if the system's response time will meet the performance requirement. Event sequence analysis states that the total CPU time for the tasks in the event sequence (C_e) is the sum of CPU time for all the tasks and the context-switching overhead. Additionally, it is also important to consider the tasks that could be executing during the time when the system responds to the lever (C_a). The total CPU time required is calculated by $C_t = C_e + C_a$[12].

In the case study, C_e is computed by summing the times for each step captured in the platform specific performance concurrent communication diagram, a portion of which is shown in Figure 6. The total time for each step is the sum of its PAdemand, PAdelay, and PAexOp. A complete listing of all the steps is shown in Table 2 rows one though nine. Additionally, there are four possible tasks that support the acceleration event; therefore, there is a minimum of four context switches used in the calculation. The context switching information can be gathered from the <<PAhost>>'s PActxtSwT tag in the deployment diagrams. In this example, all the tasks occur on the Auto Control Node so all context switching will take 1.0 milliseconds and it is multiplied by four, as shown in Table 2 row ten. All then values are then summed and the total C_e is shown in Table 2 row 11.

Table 2. CPU Time for Accelerating Event

Event Sequence (C_e)			
No.	**Step Name**	**Stereotype/Tag**	**Value**
1	Cruise Control Interrupt	<<PAstep>>/PAdemand	1.5ms
2	Cruise Control Lever Interface reads input	<<PAstep>>/PAdemand	3ms
3	Cruise control request sent	<<PAstep>>/PAdemand	1ms
4	Cruise Control receives message & changes state	<<PAstep>>/PAdemand	6.5ms
5	Cruise control command	<<PAstep>>/PAdemand	1ms
6	Speed Adjust executes command	<<PAstep>>/PAdemand	14ms
7	Speed Adjust sends throttle value	<<PAstep>>/PAdemand	1ms
8	Throttle Interface computes new position	<<PAstep>>/PAdemand	6ms
9	Throttle outputs throttle position to throttle	<<PAstep>>/PAdemand	1ms
10	Auto Control Node Context Switching	<<PAhost>>/PActxtSwT	1 ms*4
11		**Total Ce**	**39ms**

Next, C_a is computed for any overhead processing associated with this event. In this example, the events that can interrupt the accelerating event are the periodic auto sensors event and the periodic shaft rotation event. The auto activates every 100 milliseconds, which is captured on its communication diagram in the <<PAopenload>> tag. This task could therefore occur two times during the 250 milliseconds requirement; therefore the CPU time is multiplied by two as shown in Table 3 row one. Additionally, the context switching time also needs to be taken in account. This information can again be taken from the <<PAhost>>'s PActxtSwT tag from the deployment diagram, and it is shown in Table 3 row two. The second potential interrupt is the shaft rotation event that occurs every five milliseconds, and therefore it could potentially occur 50 times during the 250 milliseconds requirement period. So in the overhead and CPU time associated with a shaft rotation event are multiplied by 50 [11] as shown Table 3 rows three and four. All the times are them summed to get the total C_a for this scenario.

Table 3. Potential Interrupts for Accelerating Event

Other Task Executing (C_a)			
No.	**Task**	**Stereotype/Tag**	**Value**
1	Auto Sensors	<<PAstep>>/PArespTime	15ms*2
2	Auto Sensors context switch	<<PAhost>>/PActxtSwT	1ms *2*2
3	Shaft Interface	<<PAstep>>/PArespTime	1.75ms*50
4	Shaft Interface context switch	<<PAhost>>/PActxtSwT	1ms *2*50
5		**Total Ca**	**221.5ms**

After Ce and Ca are calculated, they are then summed to get the total time of 260.5 milliseconds. In this case, the performance requirement of 250 milliseconds is not met; therefore, changes must be made to the design, architecture, or performance requirements. This particular problem could be resolved by simply using a more powerful host on the Auto Control Node. A more powerful machine with a context switch delay of 0.5milliseconds can reduce the total time to 206.5milliseconds, which meets the performance requirement. However, in an economical car the additional cost of the powerful hardware may make the return on investment for meeting this requirement too low. Therefore other solutions such as, deploying the shaft interface on another node to reduce the processing load, could be explored to find a more economical solution. By successfully identifying the problem during design, the appropriate changes can be implemented more easily than during later in the implementation phase.

This example only illustrates how the proposed method is used for event sequence analysis. However, the proposed method can be easily used for conducting rate monotonic analysis (RMA). This is accomplished by using the UML SPT schedulability subprofile, which caters toward schedulability analysis and following the RMA approach described in [12].

4 Conclusion

Software performance is a complex issue that is a dependent on many aspects of a software system. Too often performance requirements are not addressed until after the application is developed and sometimes even after it is deployed. Failure to meet performance requirements can result in catastrophic failures and negatively impact schedule, budget, and software quality. These problems can be greatly reduced if performance requirements are modeled, analyzed, addressed, and resolved early in the software development lifecycle.

Ensuring that software product line performance needs are met is particularly important since one SPL design will potentially result in many SPL members. The approach and guidelines outlined in this paper provide a practical approach to solving this problem in SPL. The Object-Oriented Analysis and Design for Product Lines (PLUS) method provides a sound basis for modeling SPL. PLUS combined with the benefits of the UML Profile for Schedulability, Performance, and, Time (SPT), is an integrated method for capturing performance requirements in a structured manner that can later be analyzed. The performance requirements and variability can be captured in the requirements model through use cases and feature modeling. The performance requirements can then be carried through the architectural modeling by exploiting the concurrent communication diagrams to create performance contexts for the various performance scenarios. A set of common SPL member configurations can be captured and analyzed in the SPL design.

Performance analysis can be carried out on a kernel system design, consisting of kernel SPL components and any default components. However, performance analysis also needs to be carried out on each product line member design executing on a given hardware configuration, before deployment. Platform specific diagrams can be created during application engineering on a SPL member-by-SPL member basis, where they can be properly tagged with the resource information. The completed platform specific performance model can then be quantitatively analyzed to assess if it is meeting the performance requirements. If the performance requirements are not met, then changes can be made to the design, architecture, or requirements if necessary.

The entire process helps provide confidence in the design and its ability to meet performance requirements. By following the performance modeling approach taken in this paper, future software product line developments can avoid the costly problems associated with poor performance found in development and deployment.

5 References

[1] "The UML Profile for Schedulability, Performance and Time," Object Management Group (OMG) Version 1.1, January 2005.

[2] H. Harreld, "NASA Delays Satellite Launch After Finding Bugs in Software Program," vol. 2005: Federal Computer Week, 1998, pp. http://www.fcw.com/fcw/articles/1998/FCW_042098_338.asp.

[3] Finkelstein and J. Dowel, "A Comedy of Errors: the London Ambulance Service case study," presented at 8th International Workshop on Software Specification and Design (IWSSD '96), 1996.

[4] W. He and S. Goddard, "Capturing an Application's Temporal Properties with UML for Real-Time," Lincoln, Nebraska 2000.

[5] "Introduction to HRT UML," Intecs Brainware Company October 20 2004.

[6] G. Martin, L. Lavagno, and J. Louis-Guerin, "Embedded UML: a merger of real-time UML and co-design," *ACM*, 2001.

[7] S. Gerard, N. Voros, C. Koulamas, and F. Terrier, "Efficient System Modeling of Complex Real-Time Industrial Networks Using the ACCORD/UML Methodology," presented at IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems, 2000.

[8] A. Burns and A. J. Wellings, "HRT-HOOD: A Structured Design Method for Hard Real-time Systems," *Real-time Systems Journal*.

[9] X. Wu, D. McMullan, and M. Woodside, "Component Based Performance Prediction," presented at 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, Portland, Oregon, 2003.

[10] X. Wu and M. Woodside, "Performance modeling from software components," presented at Proceedings of the 4th international workshop on Software and performance, Redwood Shores, California, 2004.

[11] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley Object Technology Series, 2005.

[12] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Boston: Addison-Wesley Object Technology Series, 2000.

[13] C. Smith and L. Williams, *Performance Solutions A Practical Guide to Creating Responsive, Scalable Software*. Boston: Addison-Wesley, 2002.

Concurrency and Real-time Specification in UML

K. Lano, K. Androutsopolous, D. Clark

Dept. of Computer Science, King's College London, Strand, London, WC2R 2LS, UK

Abstract. This paper describes techniques for the specification of concurrent, real-time and distributed object systems in UML, using extensions of the OCL language.

1 Introduction

UML contains some mechanisms for real-time and concurrency modelling, such as time triggers for state machines, and the OCL *OclMessage* type. However these mechanisms are not complete. For example whilst *OclMessage* represents messages sent from an object, there is no specification facility within OCL to represent or express properties of messages *received* by an object or the temporal and synchronisation properties of the operations invoked by these messages.

In this paper we describe a coherent set of concurrency and real-time specification mechanisms for the UML-RSDS subset of UML. UML-RSDS (Reactive System Design Support) is a precise subset of UML, which provides a tool-supported process for MDD and the MDA. The emphasis in UML-RSDS is on the abstract declarative specification of systems, using platform-independent models (PIMs) which consist of class diagrams, constraints and statecharts. The declarative specifications can be used to guide and determine a choice of implementation which satisfies, *by construction*, the specification. System analysis can be performed at the PIM level by means of translations to the B [1] and SMV [5] notations, and the use of tools for these languages. Figure 1 shows the UML-RSDS development process.

For the specification of systems which involve concurrent execution of operations, the same UML notations can be used, together with notational extensions to specify particular concurrency constraints such as mutual exclusion, one-writer, many-readers, etc.

For the specification of real-time properties such as maximum possible delays in requested operations initiating execution, and the duration of operation executions, we propose the use of real-time logic [10] as an extension of OCL 2.0 [18].

Specification of distributed object systems and distributed computations can also be supported, using a declarative non-distributed specification of the effect of the computation, eg, by means of a pre and post-condition, and then applying a pattern for the coordination of distributed computations [8].

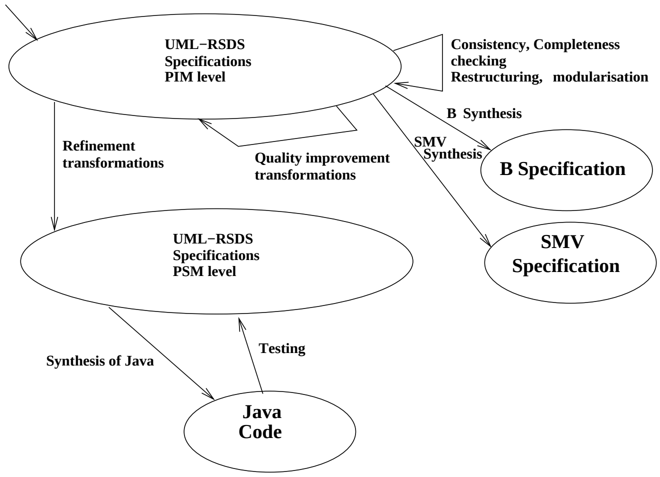


Fig. 1. UML-RSDS Process Steps

2 UML-RSDS Specifications

UML-RSDS was introduced in [14, 15] as a means of defining complex reactive systems in a concise and abstract manner [11]. It has since been extended for use with general software systems, including internet systems, for which a specialised code generator, UML2Web [16] is provided.

An example of such a specification for a lift system is shown in Figure 2. For each lift there are sensors (marked with the stereotype ?) *dest* indicating the target floor, *fps* giving the current lift position, *dcs* indicating if the lift doors are

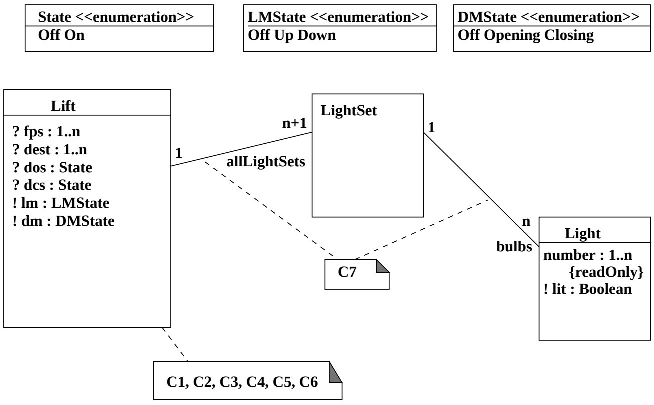


Fig. 2. Lift Control System Generic Class Diagram

closed, and *dos* indicating if the doors are open. There are actuators (stereotype !) *lm*, the lift motor, and *dm*, the door motor. If there are n floors, there are $n+1$ light sets indicating the position of the lift: one on each floor and one inside the lift. Each light set has n lights.

The constraints for the control system are:

C1 "If the door is open, or not closed, the lift motor is off":

$$dos = On \;\Rightarrow\; lm = Off$$
$$dcs = Off \;\Rightarrow\; lm = Off$$

C2 "If the destination is below the current floor, and the doors are closed, the lift motor is in the *Down* state":

$$dest < fps \;\&\; dos = Off \;\&\; dcs = On \;\Rightarrow\; lm = Down$$

C3 "If the destination is above the current floor, and the doors are closed, the lift motor is in the *Up* state":

$$dest > fps \;\&\; dos = Off \;\&\; dcs = On \;\Rightarrow\; lm = Up$$

C4 "If the lift is at the target floor, its motor is off":

$$dest = fps \;\Rightarrow\; lm = Off$$

C5 "If the lift is at the target floor, the door opens":

$$dest = fps \;\&\; dos = Off \;\Rightarrow\; dm = Opening$$
$$dest = fps \;\&\; dos = On \;\Rightarrow\; dm = Off$$

C6 "If the lift is not at the target floor, the door closes":

$$dest \;/= fps \;\&\; dcs = Off \;\Rightarrow\; dm = Closing$$
$$dest \;/= fps \;\&\; dcs = On \;\Rightarrow\; dm = Off$$

C7 "The lights corresponding to the current floor are lit":

$$fps = number \;\Rightarrow\; lit = true$$
$$fps \;/= number \;\Rightarrow\; lit = false$$

$C1$, ..., $C6$ are local invariants of the *Lift* class, $C7$ is a constraint attached to the *Lift_LightSet* and *LightSet_Light* associations.

Constraints can also be attached to operations as pre and post conditions.

From such specifications, the UML-RSDS tools can be used to:

1. Check the consistency and completeness of the specification (eg, that two constraints do not contradict each other).
2. Automatically generate executable Java code together with an API for its use as a component in an application. The code is constructed using the constraints of the system to derive the definition of operations.

3. Automatically generate B AMN [1] or SMV specifications, which can be used to carry out semantic analysis and proof on the specification, and to animate it.

An extract of the generated Java code for the *Lift* class above is:

```
public void setfps(int fpsx)
{ if (dest < fpsx && dos == Off && dcs == On) { lm = Down; }
  if (dest > fpsx && dos == Off && dcs == On) { lm = Up; }
  if (dest == fpsx) { lm = Off; }
  if (dest == fpsx && dos == Off) { dm = Opening; }
  if (dest == fpsx && dos == On) { dm = Off; }
  if (dest != fpsx && dcs == Off) { dm = Closing; }
  if (dest != fpsx && dcs == On) { dm = Off; }
}
```

This shows how the *Lift* constraints concerning *fps* are enforced.

A validation property for the lift system is:

$$lm = On \;\Rightarrow\; dm = Off$$

This can be proved true as an assertion in the B model derived from the *Lift* class.

Scheduling can be added to the lift system in a modular manner: requirements of the form

$$setreq(f) \;\Rightarrow\; AF(fps = f \;\&\; dos = On)$$

"if the lift is requested to visit floor f then eventually it will be at that floor with its door open" can be ensured by the definition of a global *Schedular* class, which intercepts *setreq* events, maintains a queue of waiting requests, and sends a command (in this case $setdest(f)$) to the lift controller when a request is due to be processed. A standard strategy for construction of *Schedular* can be used [7], with variations in the scheduling strategy expressed in the definitions of queing and dequeing operations. The controller definition is completely independent of the schedular.

3 State Machines

State machines are the key means of specifying the dynamic behaviour of objects in UML. In UML and in UML-RSDS, state machines are of two kinds:

- Protocol state machines, used to express the intended permissable sequences of operation executions which an object can undergo.
- Behaviour state machines, used to express detailed behaviour, such as the execution steps of an operation.

For state machines describing the *classifierBehavior* of (objects of) a class, the class invariants should be true in every state. For behaviour state machines attached to an operation, only the initial and terminal states need to satisfy the class invariant, unless the operation permits interruption by other operation invocations on the same object, in which case every state must satisfy the invariant.

From a protocol state machine we can:

- Check the completeness and consistency of the state machine. *Completeness* is the property that all cases of behaviour are explicitly specified on the state machine: for each state, and for each operation that the object can execute, the disjunction of guards on the transitions for that operation from the state is equivalent to *true*. *Consistency* is the property that no conflicting behaviours are specified: no two transitions from the same state for the same operation have overlapping guards. Transition actions should also be consistent with their target state invariants.
- Generate Java or B code for the operations of the class, additional to the effects of each operation specified in its postcondition.
- Check that temporal properties, expressed in CTL, are valid for the state machine [2].
- Verify behavioural compatibility of a subclass against its superclass.

3.1 Behavioural Compatibility

Behavioural compatibility refers to the requirement that the protocol defined for a superclass object should not be violated by a subclass object. Informally this consists of two conditions on the subclass state machine C and the superclass state machine A [12, 13]:

1. *Refinement*: For every state s of C, there is a state $\sigma(s)$ of A, and for every transition tr of C there is a transition $\sigma(tr)$ of A such that:
   (a) $\sigma(s)$ is initial in A if s is initial in C.
   (b) $\sigma(tr) : \sigma(s) \rightarrow \sigma(t)$ in A if $tr : s \rightarrow t$ in C.
   (c) tr and $\sigma(tr)$ have the same trigger event.
   This means that any behaviour of C must also be one of A.
2. *Adequacy*: For each state s of A there is at least one state s' of C such that $\sigma(s') = s$. The disjunction of the state invariants of all such s' is equivalent to the state invariant of s.
   For each transition $tr : s \rightarrow t$ in A there are transitions $tr' : s' \rightarrow t'$ of C such that $\sigma(tr') = tr$, for every state s' such that $\sigma(s') = s$. The disjunction of guards of the tr' should be equivalent to the guard of tr.
   This means that behaviour defined in the superclass must also be defined in the subclass.

Conditions 1 and 2 can be formally deduced from the requirement that the semantics of C, as a temporal logic theory Γ_C, is a theory extension of the theory Γ_A of A [12].

An example of behavioural compatibility is when a class is formed as an amalgamation of two other classes (Figure 3). If A and B have no common operations, and there are no new constraints on behaviour in C, then C is behaviourally compatible with A and B. This corresponds to the case of UML state redefinition where a composite state is extended by adding a region ([19], page 534).

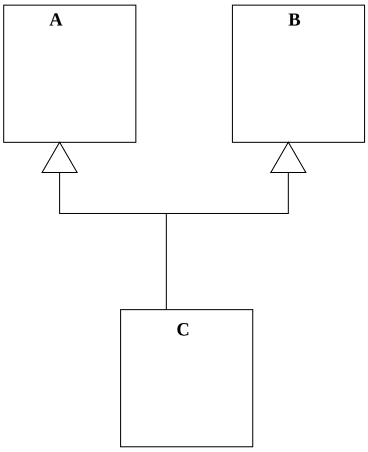


Fig. 3. Amalgamation of classes

The UML-RSDS tool automatically tries to construct a refinement mapping σ from a subclass state machine C to a superclass state machine A. If such a mapping exists it will then check it for the adequacy condition.

3.2 Operation Behaviour Specification

The detailed execution steps of an operation m can be specified by a behaviour state machine in which all of the transitions have completion event triggers. Figure 4 shows a simple example of bubblesort, for three elements a, b and c. The class invariant should hold at all initial and terminal states of the state machine, the precondition of the operation should hold at the initial state and the postcondition at all terminal states. If the operation may be interrupted by executions of other operation invocations on the same object then the class invariant should hold in all states.

These diagrams are used to:

- Synthesise sequential Java code of the operation.
- Synthesise the B implementation code of the operation, given suitable loop invariants and variants. This enables verification of the correctness of the loop with respect to an abstract pre-post specification.

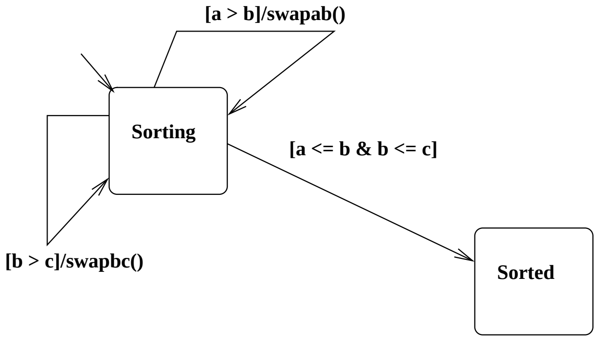


Fig. 4. Example behaviour state machine diagram

- If the class is stereotyped as ≪ *active* ≫, to synthesise the thread *run* method for a concurrent Java implementation of the operation. For simplicity, we consider that active classes only have a single thread of control per object [21].

Process algebra expressions of an operation could also be derived from such state machines.

In the first case the behaviour state machine SC attached to an operation op of a class C is used to define an explicit sequential algorithm for op as follows:

1. The set of states is represented as a new enumerated type $State_{SC}$.
2. A new attribute *op_state* of this type is added to C, together with the initialisation $op\_state = initial_{SC}$ of this attribute to the initial state of SC.
3. Any attribute of the state machine becomes a local variable of the operation.
4. The state machine yields the operational definition

   ```
   public void op(PT p)
   { $Code_{op}$ }
   ```

   where $Code_{op}$ is:

   ```
   $entry_{initial_{SC}}$;
   $op\_state = initial_{SC}$;
   while ($op\_state$ != $terms_1$ && ... &&
           $op\_state$ != $terms_m$)
   { if ($op\_state$ == $ss_1$ && $G'_1$)
     then
     { $actl_1$;  $entry_{ts_1}$;  $op\_state = ts_1$;  }
     else if ...
     else if ($op\_state$ == $ss_k$ && $G'_k$)
     then
     { $actl_k$;  $entry_{ts_k}$;  $op\_state = ts_k$;  }
   }
   ```

where the $terms_i$ are all the terminal states of SC (ie, states with no outgoing transitions), and the transitions of SC are $ss_1 \rightarrow_{[G_1]/act_1} ts_1$ upto $ss_k \rightarrow_{[G_k]/act_k} ts_k$. $actl_i$ expresses act_i in Java, using ; as the connector between individual actions.

For the bubblesort specification, the generated Java is therefore:

```
public void bsort()
{ bsort_state = sorting;
  while (bsort_state != sorted)
  { if (bsort_state == sorting && a > b)
    { swapab(); }
    else if (bsort_state == sorting && b > c)
    { swapbc(); }
    else if (bsort_state == sorting && a <= b && b <= c)
    { bsort_state = sorted; }
  }
}
```

The UML-RSDS tool can also refactor and simplify code, when the program structure is relatively simple (no nested loops or unstructured code).

In the third case the state machine is being treated as an *action system* in the sense of [3]. The transitions define the individual sequential steps which an object can take, ie, the level of granularity of the object. In contrast to [21], we assume that the transition actions cannot be interrupted, the reason for this is that states in the state machine are considered to be the only points at which the state of the object can be safely observed – ie, at which the class invariant necessarily holds.

The thread code of a behaviour state machine SC attached to an operation *run* of an active class C is generated as follows:

1. The set of states is represented as a new enumerated type $State_{SC}$.
2. A new attribute *run_state* of this type is added to C, together with the initialisation $run\_state = initial_{SC}$ of this attribute to the initial state of SC.
3. Any attribute of the state machine becomes a new attribute of the class.
4. The state machine yields the definition

```
private synchronized void run_step()
{ if ($run\_state$ == $ss_1$ && $G'_1$)
  { $actl_1$; $entry_{ts_1}$; $run\_state$ = $ts_1$; }
  else if ...
  else if ($run\_state$ == $ss_k$ && $G'_k$)
  { $actl_k$; $entry_{ts_k}$; $run\_state$ = $ts_k$; }
}
```

where the transitions of SC are $ss_1 \rightarrow_{[G_1]/act_1} ts_1$ upto $ss_k \rightarrow_{[G_k]/act_k} ts_k$. $actl_i$ expresses act_i in Java, using ; as the connector between individual actions.

The *run* method itself is defined as:

```
public void run()
{ $entry_{initial_{SC}}$;
  $run\_state \;=\; initial_{SC}$;
  while ($run\_state$ != $terms_1$ && ... && $run\_state$ != $terms_m$)
  { $run\_step()$; }
}
```

where the $terms_i$ are all the terminal states of SC.

4 Case Study: Sudoku Solver

We give an example of the generation of concurrent code from a behavioural state machine, using a simple application which fills in a partially completed Sudoku board until it is complete. The algorithm is distributed, with each individual square on the board independently checking to see if it is forced to be a particular value (ie, the list of possible values placable on the square has size one). Figure 5 shows the specification of the system. We use a variant concrete syntax of OCL, which makes constraints more concise. For example $set{\rightarrow}select(P)$ in OCL is written as $set \mid P$ in UML-RSDS.

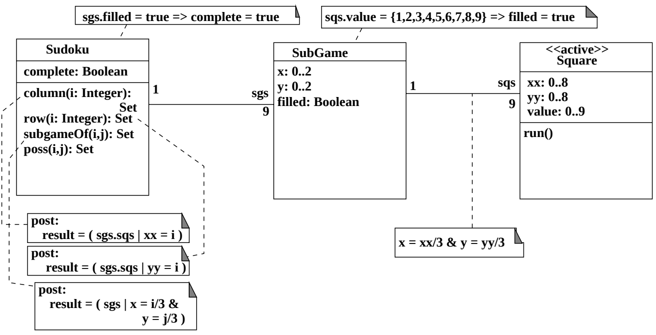


Fig. 5. Specification of Sudoku solver

The set $poss(i,j)$ of possible values for square i, j is

$$\{1,2,3,4,5,6,7,8,9\} - row(j).value - column(i).value - subgameOf(i,j).sqs.value$$

The set 1..9 minus the set of values already on the subgame containing i, j, and minus the set of values already on row j and column i.

Figure 6 shows the specification of the squares behaviour as active objects.

The corresponding Java code in the *Square* class is:

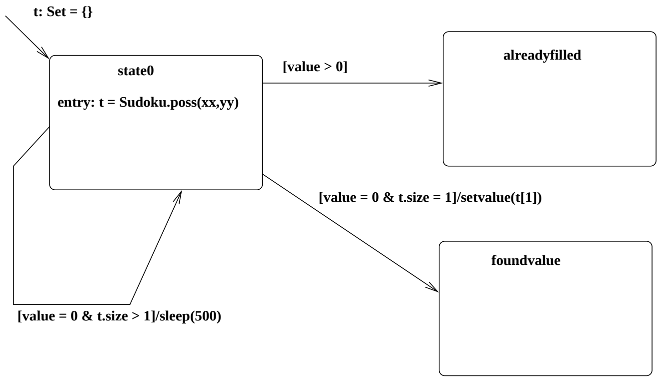

t: Set = {}
state0
entry: t = Sudoku.poss(xx,yy)
[value > 0]
alreadyfilled
[value = 0 & t.size = 1]/setvalue(t[1])
foundvalue
[value = 0 & t.size > 1]/sleep(500)

Fig. 6. Statechart of *run* method of *Square*

```
  private synchronized void run_step() {
    if (value > 0 && run_state == state0)
    { run_state = alreadyfilled; }
    else if (value == 0 && t.size() == 1 && run_state == state0)
    { Controller.inst().setvalue(this,
                       ((Integer) t.get(0)).intValue());
      run_state = foundvalue;
    }
    else if (value == 0 && t.size() > 1 && run_state == state0)
    { try { Thread.sleep(500); } catch (Exception e) { }
      t = Sudoku.poss(xx,yy);
      run_state = state0;
    }
 }

  public void run()
  { t = Sudoku.poss(xx,yy);
    run_state = state0;
    while (run_state != Square.foundvalue &&
           run_state != Square.alreadyfilled)
    { run_step(); }
  }
```

In common with OCL, normal class invariants in UML-RSDS are required to be true only at time points where the class state is observable: it is permitted to temporarily break an invariant during operation execution, for example, provided the invariant is re-established at termination of the operation. However, in objects which permit concurrent execution of operations, object state may be observed at points during an operation execution, and stronger restrictions are needed to ensure that invariants are always true at each observable time point. The following strategy is used to manage multiple threads in the concurrent implementation of an operation:

- If an action $obj.op(p)$ requires additional actions *acts* to occur, to maintain invariants, then these actions must be carried out without interruption by other threads, in the same atomic step as the update $obj.op(p)$.
  In the above example, the constraints defining the values of *filled* and *complete* need to be maintained by the *Controller* component when any *value* changes – the update code which does this is performed indivisibly, as part of the same transition action, with the update of *value*.
- If an attribute is shared for reading or writing between several threads, then its *get* and *set* operations must be atomic and uninterruptible, ie, `synchronized` in Java.

5 Concurrency Specification Notations

In order to specify properties such as mutual exclusion between operations, we adopt the event counters of VDM^{++}, which are given a formal semantics in [10]:

1. $\#req(m)$ – the number of requests received up to and including the present time for operation m
2. $\#act(m)$ – the number of execution initiations of m so far
3. $\#fin(m)$ – the number of execution terminations of m so far.

Thus

$$\#waiting(m) \;=\; \#req(m) - \#act(m)$$

denotes the number of waiting executions of m, and

$$\#active(m) \;=\; \#act(m) - \#fin(m)$$

is the number of currently executing instances of m. All of these event counters act like integer-valued instance-scope attributes of the classes in which m is defined. They relate directly to the informal semantics of UML, for example $\#waiting(m)$ is the number of call event instances of m in the input pool of the object [19].

To specify that an operation is purely sequential and cannot be executed concurrently with itself on the same object, we can write:

$$\#active(m) \;\leq\; 1$$

as an invariant of the class. This could be ensured by implementing m as a *synchronized* method in Java. To express the stronger property that no two invocations of the operation can be concurrently executing, even on different objects, we can write:

$$C.allInstances().\#active(m).sum \;\leq\; 0$$

To express mutual exclusion between two operations m and n, the invariant

$$\#active(m) + \#active(n) \;\leq\; 1$$

can be written.

Another form of concurrency specification is the *permission predicate* of VDM^{++}. This defines a guard on an operation, a condition which must be true for the operation to execute. For example:

$$get() \;\Rightarrow\; buffer.size > 0$$

for a shared buffer. Such a condition is stronger than a precondition, implying blocking semantics if a caller attempts to execute the operation when the condition is false. It is usually preferable to express guards using event counters instead of instance variables, for greater abstraction. For the buffer, we could write:

$$get() \;\Rightarrow\; \#fin(put) > \#act(get)$$

In general a wide range of concurrency properties can be expressed using event counters. However in some cases, eg, to specify that the waiting operation instance with the highest priority (measured on the basis of its parameters or its arrival time) should execute before any other instance of the operation, a more elaborate semantic model and constraint language is necessary. In UML, each operation is also a class [19], and its instances represent individual executions of the operation (Figure 7). For each such instance, say mx, we can consider that there are attribute values $mx.p$ for each formal input parameter p of m. For real-time specification we will also consider that the start, end and request time of the execution are recorded as attibutes of mx, together with the time that the request which caused this execution occurred, and the invoker of the behaviour. Instances of M come into existence at the point where an execution of m is requested via an operation call action. This model provides similar facilities to the *OclMessage* type in OCL [18]. The OCL operators $^\frown$ and $^{\frown\frown}$ on messages can be defined in terms of the model.

The event counters and other simple forms of property can be considered as convenient abbreviations for constraints expressed using the model of Figure 7. Table 1 shows how event counters are defined in the extended model. M is the class corresponding to operation m, and the *host* role of M is the object on which m is executing or which is the intended executor (target) of m.

Abbreviated formula	*Meaning*
#req(m)	$(M \mid req\_time \leq now \;\&\; host = self).size$
#act(m)	$(M \mid act\_time \leq now \;\&\; host = self).size$
#fin(m)	$(M \mid fin\_time \leq now \;\&\; host = self).size$
#active(m)	#act(m) - #fin(m)
#waiting(m)	#req(m) - #act(m)

Table 1. Semantics of event counters

Implementation of concurrency properties in a specific platform such as Java can be carried out by the definition of variables implementing event counters, and

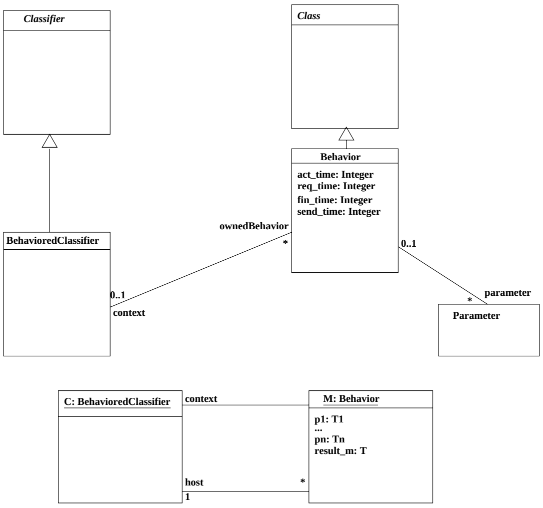


Fig. 7. Conceptual execution semantics model

the use of guard expressions preventing operation execution if a required property would be violated. Only certain properties can be ensured in this manner, however. In particular, since the request queue for methods cannot be manipulated directly in Java, constraints involving $\#req(m)$ and $\#waiting(m)$ cannot always be ensured.

6 Code Generation for Concurrency Specifications

The UML-RSDS development process uses automated generation of code from high-level models, using various strategies to generate code which satisfies, by construction, the constraints of the specification, without the need of further verification. This approach can also be applied for systems involving specification of concurrency properties.

The benefits of automated code generation are particularly significant when applications must use a complex platform such as J2EE, or complex programming mechanisms such as multi-threading. By using a systematic automated process for the implementation of constraints in code, code generation removes the possibility of human errors caused by insufficient understanding of the platform or programming mechanisms.

Three patterns [6] are used for the automated implementation of particular forms of concurrency constraints in Java:

- *Mutex objects*
- *Guarded methods*
- *Synchroniser objects*

Mutex objects are objects in which every operation is synchronized. Classes with the stereotype 'sequential' are translated into such classes.

If there are permission constraints of the form

$$op(pars) \;\Rightarrow\; G$$

then *op* can be defined as a guarded operation, for example:

```
class GBuffer
{ ArrayList buffer = new ArrayList();
  ...
  synchronized String get()
  { while (buffer.size() == 0)
    { try { wait(); }  /* Wait until buffer.size() > 0 */
      catch(InterruptedException e) { } }
    return (String) buffer.get(0);
  }
}
```

in the case of a buffer.

Synchroniser objects are used to implement general invariants involving event counters. Variables recording the values of the counters are defined in the synchroniser object, and the operations of this object are invoked by the objects whose concurrency is being managed by the synchroniser. It can therefore enforce invariants involving event counters, by preventing executions of operations which would violate the invariants. Figure 8 shows the general structure of the pattern. It is a generalisation of the concept of a semaphore.

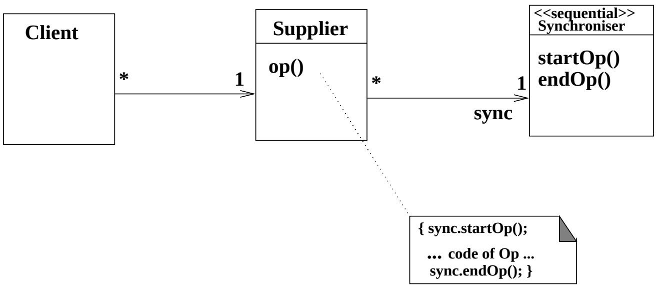


Fig. 8. Class Diagram of Synchroniser Object Pattern

Given a set of invariants *Inv* for the concurrent behaviour of class C, expressed in terms of event counters for the operations of C, we:

- Identify the 'preconditions' $[act(op)]Inv$ and $[fin(op)]Inv$ of each of the initiation and termination events of the operations of C with respect to *Inv*. If a precondition is not implied by *Inv*, then a guard condition must be placed

on the corresponding *startOp* or *endOp* operation of the synchroniser object, to block this operation (and the caller) until the precondition is true.
If an event may make a precondition true, then the corresponding *start* or *end* operation should have a *notify* invocation at its termination, to release any callers which are waiting on the precondition.

For example, consider a specification of a readers/writers protocol on an object:

```
class RW implements Runnable
{ int data;

  int read1()
  { return data; }

  void write1()
  { data = data*5; }
}
```

with the invariants *Inv*:

$$\#active(write1) \leq 1$$
$$\#active(write1) > 0 \;\Rightarrow\; \#active(read1) = 0$$

A suitable synchroniser object is:

```
class RWSynchroniser
{ int writers = 0;  /* Sum of #active(w) for all writers w */
  int readers = 0;  /* Sum of #active(r) for all readers r */
  /* Invariant: (writers > 0  =>  readers = 0) &           */
  /*            writers <= 1                               */

  synchronized public void startRead()
  { while (writers > 0)
    { try { wait(); } catch (InterruptedException e) {} }
    readers++;
  }

  synchronized public void startWrite()
  { while (writers > 0 ||  readers > 0)
    { try { wait(); } catch (InterruptedException e) {} }
    writers++;
  }

  synchronized public void endRead()
  { readers--;
    notifyAll();
  }

  synchronized public void endWrite()
  { writers--;
```

```
    notifyAll();
  }
}
```

The operations are derived mechanically, using the precondition calculation

$$[act(read1)]Inv$$

to compute the guard

$$\begin{aligned}&[readers++](writers > 0 \;\Rightarrow\; readers = 0) \;\equiv\\ &(writers > 0 \;\Rightarrow\; false) \;\equiv\\ &writers = 0\end{aligned}$$

of *startRead* and likewise for *startWrite*.

The *RW* class with the synchronisation object is:

```
class RW implements Runnable
{ int data;
  RWSynchroniser rwsync;

  int read1()
  { rwsync.startRead();
    int local = data;
    rwsync.endRead();
    return local;
  }

  void write1()
  { rwsync.startWrite();
    data = data*5;
    rwsync.endWrite();
  }
}
```

endOp operations should also be invoked if *op* terminates abnormally because of an exception, ie, these should be in the *finally* clause of a suitable *try* statement within the code of *op*.

A single synchroniser object can be shared between several objects, to restrict inter-object concurrency as well as intra-object concurrency. The formal correctness of this implementation strategy is shown in [10].

7 Real-time Specification

The UML profile for schedulability, performance and time [20] uses the key concept of a *stimulus* to tie together a request for a supplier object service, the reception of this request and its processing.

We will use the following notations of Real-time action logic (RAL) [10] for specific times in the history of a stimulus:

- $\leftarrow(i)$ – the time of the stimulus generation event which is the cause of the stimulus i.
- $\rightarrow(i)$ – the time of the stimulus reception event whose cause is i. This time might not be defined (its value is *OclInvalid*) if the stimulus is lost in transit. In terms of the UML 2.0 superstructure ([19], page 420), this corresponds to the time that "an event occurrence is recognized by an object that is an instance of a behaviored classifier". The event occurrence i may be placed in an input pool/queue for later processing, or immediately processed
- $\uparrow(i)$ – time of the *ScenarioStartEvent* for the scenario execution whose cause is the stimulus reception of i. Typically this is the start of an execution of an operation m with parameters *pars* on a target object. This time may be undefined if the request is never scheduled for execution. It is the time that "a behavior is invoked as determined by the event" ([19], page 420).
- $\downarrow(i)$ – time of the *ScenarioEndEvent* for the execution triggered by i.

In a sequence diagram these can be graphically represented as shown in Figure 9.

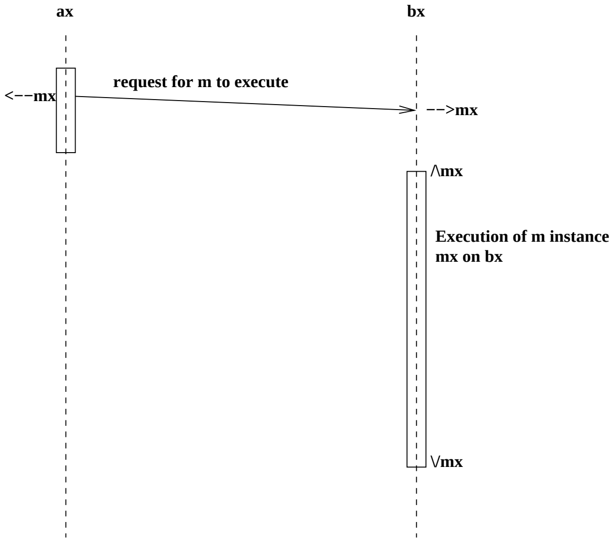


Fig. 9. Example sequence diagram

Using these notations we can specify properties such as the maximum permitted delay t in responding to a request:

$$\uparrow(m) - \rightarrow(m) \ \leq \ t$$

and the maximum allowed duration T of execution of m:

$$\downarrow(m) - \uparrow(m) \ \leq \ T$$

In the semantic model (Figure 7), these constraints are interpreted as invariants of the M classes, since the times of operation events are expressed as attributes of this class, whose objects represent execution instances of m (Table 2).

Abbreviated formula	*Semantics*
$\uparrow mx$	$mx.act\_time$
$\rightarrow mx$	$mx.req\_time$
$delay(mx)$	$mx.act\_time - mx.req\_time$
$\downarrow mx$	$mx.fin\_time$
$duration(mx)$	$mx.fin\_time - mx.act\_time$
$periodic(m, \tau, \delta)$	$\forall\, i : \mathbb{N} \cdot \exists\, mx : M \cdot \tau * i - \delta \leq mx.act\_time\ \&$ $mx.act\_time \leq \tau * i + \delta$
$FIFO(m)$	$\forall\, mx1, mx2 : M \cdot mx1.req\_time < mx2.req\_time \Rightarrow$ $mx1.act\_time \leq mx2.act\_time$
$SJF(m, f)$	$\forall\, mx1, mx2 : M \cdot mx1.f < mx2.f\ \&$ $mx1.req\_time < mx2.act\_time \Rightarrow$ $mx1.act\_time \leq mx2.act\_time$

Table 2. Semantics of real-time specification notations

$periodic(m, \tau, \delta)$ means that m executes periodically, every τ time units, with a maximum permitted delay of δ. $FIFO(m)$ means that execution instances are scheduled for execution in the same order that their requests arrived at the host object. $SJF(m, f)$ means that the schedule order instead depends on the function f of the execution instances.

Ideally, constraints concerning requirements on the duration of operations could be used to guide the choice of implementation, for example, by applications of patterns such as Half-Sync/Half-Async [22] or Active Object [17]. At present however, the real-time notation described here is purely for the purposes of specification, and is not used to guide code generation.

Other forms of temporal constraints concern the duration of states: $duration(P)$ for a predicate P is the maximum length of any time interval in which P is continuously true. For example, in the lift system, we might specify that a door only attempts to open without success for at most 10 seconds:

$$duration(dm = Opening\ \&\ dos = Off) \ \leq\ 10000$$

These constraints can be ensured by using timers, if they involve at least one actuator attribute. Otherwise they represent environmental assumptions for normal behaviour of the system.

Further real-time specification capabilities, such as the *time variables* and continuous to discrete refinements of [9] could also be incorporated into UML-RSDS.

8 Verification of Concurrency and Real-time Properties

In cases where automated generation of code to satisfy concurrency and real-time constraints is not available, verification techniques to prove the correctness of the code are necessary.

In UML-RSDS, the B and SMV notations are used to provide semantic analysis of specifications. B and SMV are tools which are designed for analysis of sequential systems, where all operation executions are atomic. We can however adapt them for the analysis of concurrency and real-time properties by introducing atomic actions $act(m)$, $fin(m)$ and $req(m)$ for each update operation m, which respectively represent the activation, termination and request of m. These actions increment the corresponding counters $\#act(m)$, etc. Event counters and time attributes of the operation classes are represented as attributes in the corresponding B or SMV modules: each class C is represented by a set cs of existing objects of the class, and each operation m of C is represented by a set ms of execution instances of m, as in Figure 7.

Properties such as

$$AG(act(m) \;\Rightarrow\; AF(fin(m)))$$

"m always terminates" can be expressed and verified in SMV using this encoding of behaviour. Liveness and absence of deadlock properties can also be expressed and verified using the SMV formalism.

9 Related Work

Extensions of the VDM++ language to specify and analyse real-time distributed systems are proposed in [23]. This supports the modelling of distributed resources and communication. An operational semantics for UML state machines is defined in [21]. Our approach to UML development is similar to that of [4], which carries out performance analysis of a system specified in UML, by means of a translation to a process algebra and analysis tools for this algebra.

10 Conclusion

We have shown how the UML-RSDS method and notation can be extended to represent concurrency and real-time properties of a system. The UML-RSDS approach, of automated system generation from high-level declarative specifications, is still relevant for such extended properties, and we have described strategies for code generation to implement a range of concurrency specifications. Representation in B or SMV of concurrency and real-time properties allows reasoning about such properties, and animation of specifications.

References

1. J-R Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Androutsopoulos, K., *Verification of Reactive System Specifications using Model Checking*, PhD thesis, King's College, 2004.
3. R. Back, K. Sere, *Stepwise Refinement of Action Systems*, MPC 1989, pp. 115–138.
4. A. Bennett, A. Field, *Performance Engineering with the UML Profile for Schedulability, Performance and Time: A Case Study*, Proc. IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Volendam, Netherlands, October 2004.
5. B. Berard et al, *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer-Verlag, 1999.
6. Grand, M., *Patterns in Java, Vol. 1*, Wiley, 1998.
7. Kan, P., *Specification of Reactive Systems using RSDS*, PhD thesis, King's College London, 2006.
8. K. Lano and S. Goldsack, *Refinement of Distributed Object Systems*, FMOODS '96, Paris, 1996. Proceedings published by Chapman and Hall.
9. S. Goldsack, K. Lano and A. Sanchez, *Transforming Continuous into Discrete Specifications with VDM*$^{++}$, IEE C8 Colloquium Digest on Hybrid Control for real-time Systems, 1996.
10. Lano, K., *Logical Specification of Reactive and Real-Time Systems*, *Journal of Logic and Computation*, Vol. 8, No. 5, pp 679–711, 1998.
11. K. Lano, J. Bicarregui, P. Kan, *Experiences of using Formal Methods for Chemical Process Control Specification*, Control Engineering Practice 8 (2000), pp 71–79.
12. K. Lano, D. Clark, K. Androutsopoulos, P. Kan, *Invariant-based Synthesis of Fault-tolerant Systems*, FTRTFT 2000.
13. Lano, K., Clark, D., Androutsopoulos, K., *From Implicit Specifications to Explicit Designs in Reactive System Development*, IFM 2002.
14. Lano, K., Clark, D., Androutsopoulos, K., *Formal Specification and Verification of Railway Systems using UML*, FORMS 2003.
15. Lano, K., Clark, D., Androutsopoulos, K., *UML to B: Formal Verification of Object-oriented Models*, IFM 2004.
16. Lano, K., *Advanced System Design with Java, UML and MDA*, Elsevier, 2005.
17. R. Lavender, D. Schmidt, *Active Object: an Object Behavioral Pattern for Concurrent Programming*, Pattern Languages of Programming, 1995.
18. OMG, *UML OCL 2.0 Specification*, ptc/2005-06-06, http://www.omg.org/uml/, 2005.
19. OMG, *UML 2.0 Superstructure*, OMG Document formal/05-07-04, 2005.
20. OMG, *UML Profile for Schedulability, Performance and Time*, Version 1.1, http://www.omg.org/, 2005.
21. G. Reggio, E. Astesiano, C. Choppy, H. Hussmann, *Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach*, FASE 2000, LNCS 1783, Springer-Verlag, 2000.
22. D. Schmidt, C. Cranor, *Half-Sync/Half-Async: An architectural pattern for efficient and well-structured concurrent I/O*, Pattern Languages of Programming, 1995.
23. Verhoef, M., Larsen, P., Hooman, J., *Modelling and Validating Distributed Embedded Systems with VDM++*, Engineering College of Aarhus, Denmark, 2006.

Modeling WS-BPEL with RT-UML Diagrams [⋆]

M.Emilia Cambronero J.José Pardo Gregorio Díaz Valentín Valero

Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha
Escuela Politécnica Superior de Albacete. 02071 - SPAIN
{emicp,jpardo,gregorio,valentin}@info-ab.uclm.es

Abstract. RT-UML is a UML Profile for modeling Real Time Systems, which can be used in particular to describe Web Services Orchestration with time constraints. For that purpose, we can use two classical UML diagrams, namely, the sequence and activity diagrams. Our goal in this paper is the design of Web Services with time restriction by using RT-UML. We introduce a translation of the main elements of RT-UML into Web Services Business Process Execution Language. In a previous works we presented a translation from WS-CDL and WS-BPEL into Timed Automata, which allows us to apply verification techniques and thus, to check some properties of interest.

1 Introduction

In the last few years some new techniques and languages for developing distributed applications have appeared, such as the Extensible Markup Language, XML [1], and some new Web Services frameworks [2,3,4] for describing interoperable data and platform neutral business interfaces, enabling more open business transactions to be developed.

Web Services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web Service is an autonomous, standards-based component whose public interfaces are defined and described using XML [5,6]. Other systems may interact with a Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.

The context in which this work is situated is the development of a methodology for the generation and verification of "correct" Web Services. This metodology [7] establishes a top-down design for developing web services. The analysis phase is performed by a goal-model that specifies the properties that this kind of Real-Time systems must satisfy. Then, in the design phase we consider UML sequence and activity diagrams, in which we can capture the system time constraints. Between design and implementation we use a model-checking technique for discovering some errors and in this way, we improve the quality of design.

[⋆] This work has been supported by the CICYT project "Description and Evaluation of Distributed Systems and Application to Multimedia Systems",TIC2003-07848-C02-02, with partial FEDER support and UCLM Project "Aplicacin de Mtodos Formales al Desarrollo y Verificacin de Web Services

The source document for the implementation phase is a WS-BPEL document. Figure 1 shows this generation process. In this figure the RT-UML diagrams are translated into Web Services documents, which in their turn are also translated into Timed Automata in order to perform model checking on them.

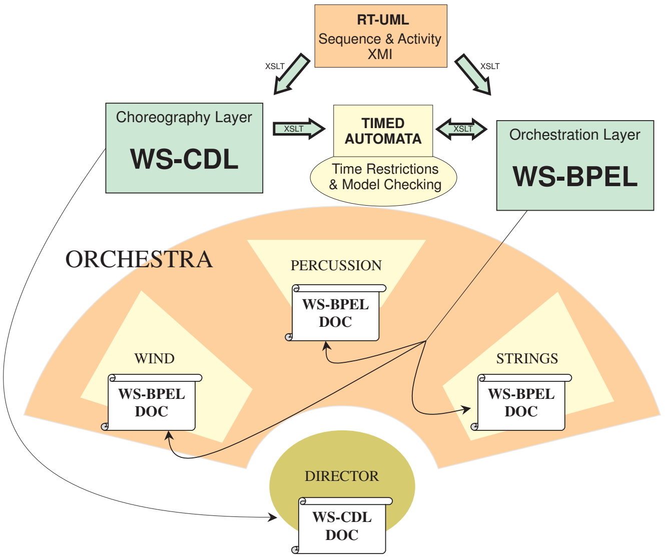


Fig. 1. The generation of correct Web Services

In [8,9,10] we have translated Web Services with time restrictions described by WS-CDL and WS-BPEL into timed automata, in order to simulate their behavior and verify some properties of interest. For that purpose, we used a well known tool that supports this formalism, UPPAAL [11] to simulate and analyze the system behavior.

We can find some related works in the literature: Mantell [12] proposes a methodology for translating UML activity diagrams into bpel4ws 1.0 Specification, and Skogan et al [13] propose another approach based on bpel4ws 1.1. In this paper we present a different vision, focusing our attention on the timed aspects of the systems. Hence, we are mainly concerned with Web Services with time constraints. It is important to highlight that we use both Activity diagrams and Sequence Diagrams, as well as the RT-UML Profile in order to model the time constraints of Real Time Web Services.

Most of the proposed composition languages for Web Services are based on XML [1], and although XML-based representations have their advantages as universal representations and exchange formats, they can be difficult to understand and to write for non-XML experts. Thus, the use of a graphical modeling language can be very useful to understand the behavior of Systems.

The motivation of this paper is the use of the Unified Modeling Language (UML) [14], and more specifically the RT-UML profile, as graphical modeling language for XML Real-Timed Web Services composition and the verification of these systems by using Model Checking techniques on Timed Automata. In other words, in this paper we focus our attention on the right part of Fig. 1, that is to say, the translation from RT-UML into WS-BPEL and from WS-BPEL into Timed Automata, this last part being shown in [8].

UML has been defined by the Object Management Group (OMG), the leading organization for object-oriented programming. RT-UML [15] allows us to see graphically the description of a Web Service business process, which helps us to understand the behavior of the participants in a business interaction; for that purpose the RT-UML diagrams are very useful, since they allow us to see this behavior in a clear way. More specifically, we use the sequence and activity diagrams for the description, since these diagrams can capture the main elements of a Web Services Business Process Execution Language and their time constraints. Afterwards, we then translate these diagrams into Web Services.

We have implemented transformation rules that can transform RT-UML models into WS-BPEL documents and these documents into Timed Automata, capturing, in every case, the time constraints. The method we propose uses RT-UML Sequence and Activity Diagrams to design Web Service compositions, and OMG's Model Driven Architecture (MDA [16]) to generate specifications in WS-BPEL and in Timed Automata.

As an illustration of this methodology, we use a particular case study, an Efficient Power Production Management System, whose description contains some time constraints.

The paper is structured as follows. In Section 2 we describe the main features of WS-BPEL. In Section 3 we present a brief description of the UML Profile for Real-time Systems, RT-UML, and we introduce the key elements of this profile, which is used in our translation to capture the System time constraints. The translation of RT-UML Sequence and Activity diagrams into WS-BPEL documents is presented in Section 4. In Section 5 we apply this methodology to the case study. Finally, the conclusions and future work are presented in Section 6.

2 WS-BPEL Description

The aim of the Web Services Business Execution Language, also known as Web Services Orchestration specification is to be able to describe precisely the behavior of any type of party and the collaboration with other parties, regardless of the supporting platform or programming model used by the implementation of the hosting environment. Using the Web Services Orchestration specification, a behavior is produced that contains a "specific" definition of the detailed ordering conditions and constraints under which behavior is performed, and which describes the specific internal behavior of the exchanged messages with all the parties involved.

In real-world scenarios, corporate entities are often unwilling to delegate control of their business processes to their integration partners. Orchestration offers a means by which the rules of participation within a collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its processes as determined by the behavior described in the Orchestration document.

We will now introduce the reader to WS-BPEL. This is an interface description language that describes the observable behavior of a service by defining business processes consisting of stateful long-running interactions, in which each interaction has a beginning, a defined behavior and an end, all of this being modeled by a flow, which consists of a sequence of activities. The behavior context of each activity is defined by a scope, which provides fault handlers, event handlers, compensation handlers, a set of data variables and correlation sets.

Let us now see a brief description of these components:

- **Events**, which describe the flow execution in an event driven manner.
- **Variables**, which are defined by using WSDL schemes, for internal or external purposes, and are used in the message flow.
- **Correlations**, which identify processes interacting by means of messages.
- **Fault handling**, defining the behavior when an exception has been thrown.
- **Event handling**, defining the behavior when an event occurs.
- **Activities**, which represent the basic unit of behavior of a Web Service. In essence, WS-BPEL describes the behavior of a Web Service in terms of choreographed activities.

3 UML Profile for Schedulability, Performance, and Time

In this section, we introduce the key elements of the profile that we have used in our translation, but first it is important to use this profile to model Real-time Systems.

The use of this profile [15] is justified because UML is lacking in some key areas that are of particular concern to real-time system designers and developers. In particular, the lack of a quantifiable notion of time and resources was an impediment to its broader use in the real-time and embedded domain. It was discovered that UML had all the requisite mechanisms for addressing these issues, in particular through its extensibility faculties. The UML Profile for Schedulability, Performance, and Time is a standard way of using these capabilities to represent concepts and practices from the real-time domain.

One of the main guiding principles is that modelers, as far as possible, should not be hindered in the way they use UML to represent their systems in order to be able to do model analysis. That is, rather than enforcing a specific approach or modeling style for real-time systems, the profile should allow modelers to choose the style and modeling constructs that they feel are the best fit for their needs of the moment.

In this paper we have chosen the "General Time Modeling" viewpoint to model Real-time Systems, because this model describes a general framework for representing time and time-related mechanisms that are appropriate for modeling real-time software systems. It is quite general, but a given application only needs to use a subset of the concepts and semantics required. We have then chosen a set of the concepts of this viewpoint to model our real-time Web Services.

The concepts of RT-UML that we have used in this paper to capture the time constraints of Web Services are the following:

- TimedAction: This very general and very useful concept is modeled by applying the <<RTaction>> stereotype to any model element that specifies an action execution or its specification (which is a way of defining defaults for instances of those specifications). This includes action executions, methods, actions (including entry and exit actions of state machines), action states, subactivity states, states, and transitions. It can also be applied to stimuli that take time to arrive at their destination.

  Timed Actions have associated the following tagged values:

  duration: The time interval in which the action occurs.
  start: The time of the event occurrence when the action started.
  end: The time of the event occurrence when the action was completed.

  The start and end times of the action are specified by appropriate tagged values (RTstart and RTend respectively). Alternatively, they may be tagged with the RTduration tag. The two forms are mutually exclusive.
- Delay: A kind of timed action execution that represents a null operation for a pre-specified time interval. This action has no side-effects except to delay the action execution that follows it. This is modeled by a model element that is stereotyped as <<RTdelay>>. It can only have an RTduration tag associated with it. Delays can be placed on the same model elements as timed actions.
- TimedEvent: This very general and very useful concept is modeled by applying the <<RTevent>> stereotype to any model element that implies an event occurrence, and it models any event that occurs at a known time instant.

  The TimedEvent only uses the tag value ”RTat”, indicating the instant at which the event occurs.
- Clock: A Clock is a kind of timing mechanism that generates a clock interrupt periodically, where a TimingMechanism is an abstract concept that captures the common features of resources that specialize in performing time measurement and timing-related functions.

  Clocks are modeled by model elements that are stereotyped as <<RTclock>>. An instance of a clock can be identified using the ”RTclockId” tag.
- reset(): An operation that stops the timing mechanism and sets it back into its initial state. The stereotype <<RTreset>> models the reset() operation on a timing mechanism.

Object	⟶Process
Messages	⟶Partnerlinktypes + portTypes + message + singleMessage
singleMessage	⟶receive | request| reply
Variables	⟶Variables
Expressions	⟶Xpath expresions | Assign activity
Activities	⟶Ordering structure Sequence$^+$ | Activity
Decision Points and Merge Construct	⟶Structured activity if
Concurrent Activities	⟶wait activity + Structured activity Flow + wait activity
Guards	⟶Condition of Activity if
Swimlane	⟶Ordering structure Sequence$^+$
RTaction	⟶Activity with scope definition
RTevent	⟶wait activity + activity
RTdelay	⟶wait activity
RTclock	⟶Variable (clock)
RTreset	⟶assign activity

Where the symbols +, | are BNF notation, and & is used to join information

Fig. 2. Schematic view of translation

4 Translation from RT-UML Diagrams into WS-BPEL

In this section we show the translation of the RT-UML sequence and activity diagrams into WS-BPEL XML format. The first subsection explains the translation for Sequence Diagrams, the second one shows the translation of activity diagram elements and the third one explains the tranlation of real-time elements. These translations have been developed with XSLT [17], XML Stylesheets Language for Transformation, which is a language for transforming XML into other XML documents. In the market we can find some tools that allow us to model the RT-UML diagrams and to obtain the corresponding XMI document, as Artisan 6.0 tool [18], this document can be easily translated to the WS-BPEL XML documents by using XSLT. In Table 2 we can see a schematic view of the translation.

4.1 Translation from RT-UML Sequence Diagrams into WS-BPEL

The system is designed by using RT-UML activity and sequence diagrams. But notice that we just need to use a single sequence diagrams, representing the whole system.

The translation of the main elements of RT-UML sequence diagrams into WS-BPEL is as follows:

- **Messages** : The messages in RT-UML Sequence diagrams correspond to the following entities of WS-BPEL:

  - PartnerLinkTypes: The partnerLinkTypes represent the interaction between a service and each of the parties with which it interacts. The main utility of PartnerLinkType is to characterize the relationship between two services, this is possible by defining the **roles** played by each of the services that participate in the conversation and specifying the **portType** provided by each service to receive messages.

    Each role specifies exactly one WSDL portType.

- **Objects** : The objects in UML Sequence diagrams can be represented by the following entities in WS-BPEL:
  - Processes: In WS-BPEL we can find two types of processes, the execution process, and the abstract processes. Execution processes model the behavior of each particular participant in the interaction, while abstract processes are meant to couple Web Service interface definitions with behavioral specifications that can be used both to constrain the implementation of business roles and to define in precise terms the behavior that each party in a business protocol can expect from the others. Then, the translation of the RT-UML objects is made only by using the execution processes, since they allow us to model the behavior of each particular participant in an interaction.

    From the RT-UML description we obtain the needed elements to complete the WS-BPEL process definition; more precisely:

    1. Variables section: here the variables identified in the RT-UML document must be declared.
    2. PartnerLinks section: in this section the different parties that interact with this particular business process are declared. This information can be easily extracted from the RT-UML document. Notice that each partnerlink must have a partnerlinktype (that models the interaction between the parties) and a role name, which is the other participant in the interaction.
    3. FaultHandlers section: where the activities that must be performed in response to faults resulting from the invocation of the assessment and aproval services. The information contained in this section is extracted from RT-UML frames, and more specifically, with the ”opt” and ”alt” labels.
    4. EventHandlers section: here the activities that must be performed in response to events or alarms are described, and this information is taken from the RTevent of RT-UML.

- **Expressions** :

  RT-UML expressions are immediately translated into WS-BPEL expressions, but the needed variables must be first declared in WS-BPEL. Notice that we should use XPath 1.0 to introduce the expressions in WS-BPEL.

  Concening with assignments, we can use the assign activities of WS-BPEL to copy data from one variable to another, to construct and insert new data using expressions.

4.2 Translation from UML Activity Diagrams into WS-BPEL

Let us now see the translation of the main elements of RT-UML activity diagrams into WS-BPEL.

- **Activities** : An activity on a UML activity diagram typically represents the invocation of an operation, a step in a business process, or an entire business process.
  Single activities are immediately translated into basic activities of WS-BPEL. More complex activities of RT-UML will be sequential (in fact, there will be an initial and a final activity), and they will be translated by using the *sequence* structured activity of WS-BPEL.
- **Decision points and Merge construct** : in RT-UML they are modeled by using diamonds. They are translated into WS-BPEL *if* structured activities. In fact, we can have an "elseif" element, and an "else" element.
- **Guards** : The guards of RT-UML are translated by using the *condition* part of the "if" activities in WS-BPEL.
- **Concurrent activities and synchronization**:
  In RT-UML Activity Diagrams the starting point of these parallel activities is represented by a horizontal split, which is called a fork, and it has one transition entering it and two or more transitions leaving it. There is also another horizontal line that represents the end of the parallel activities, with two or more transitions entering it and only one leaving it. In WS-BPEL the concurrency and synchronization between activities is provided by the *flow* Structured Activity. Within the flow structured activity we must indicate the activities that are made in parallel.
  To capture the initial synchronization among the involved activities we use a *wait* basic activity, as well as to synchronize the end of all these activities.
- **Swimlane guidelines** : This is a way to group activities performed by the same actor on an activity diagram.
  In WS-BPEL we can use the Ordering structure *Sequence to represent the Swimlane guidelines of RT-UML Activity Diagrams.*

4.3 Translation of Real-Time elements of RT-UML diagrams

RT-UML introduces three new elements in the Sequence diagrams to capture some important elements of real-time systems such as delays, events and actions.

The translation of these elements works as follows:

- **RTDelay** represents a delay for some units of time. In WS-BPEL we can use the *wait* activity to indicate this delay.
- **RTEvent** is used to indicate an event which occurs at a known time instant. To translate this element to WS-BPEL we use again a *wait* activity followed with the activity associated with RTEvent.
- **RTAction** is a timed action having a duration (alternatively you can indicate a starting and finishing time).

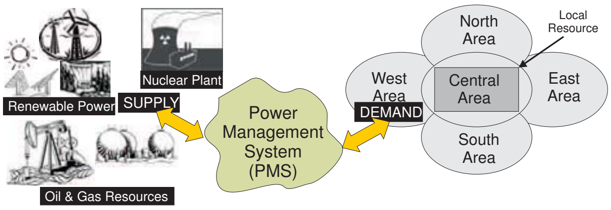


Fig. 3. The power supply versus the customer demand.

We use a *scope* element to translate this kind of action. In this scope we define an *onalarm* event, we translate the duration of the activity by using the *for* part within *onalarm*, which allows us to establish this duration.

- **RTreset** changes a clock value to zero, then it is immediately translated in WS-BPEL by using an *assing* activity.
- **RTclock** is a clock in RT-UML, and it is translated by using XPath variables.

5 Case Study: Efficient Power Production Management System

This example defines the general characteristics for an efficient power production management system with the basic restriction that the electric power is *non-storable*. This restriction implies that the power supply must be adapted to the demand. This example is based on the management system of the Spanish power lines.

In this system, we have considered three different types of electric power resources: Renewable power, pollution power and high-risk power. The renewable power plants are wind turbines, solar plants and hydroelectric plants. The pollution power plants are oil and natural gas plants. And the high-risk resources are nuclear plants.

Furthermore, the power demand is divided into areas: Central, north, south, east and west. The central area is a location where a set of resources is settled, whereas, the neighboring areas are not directly connected with electric power resources.

This system is depicted in Figures 3 and 4. The power management system described in Fig. 4 shows a pyramidal system for attending and managing the demand versus the supply. The restrictions of this system consist of two types of general restrictions and time restrictions, and are defined as follows:

- **General Restrictions**
  1. *Hierarchical demand* . Two levels for managing the demands, firstly local and secondly central.
  2. *Environmental restrictions*. If pollution environmental sensors are activated, then only renewal and high-risk power resources are available.

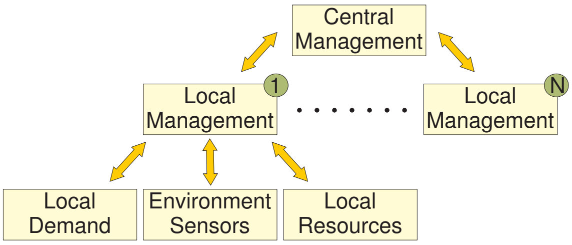


Fig. 4. The power management system (PMS).

Furthermore, these sensors inform of the amount of power that the renewal resources could supply.

3. *Hierarchical supply.* The priority for selecting the resources is: firstly renewable resources, secondly pollution resources and thirdly high-risk resources.

- **Time Restrictions**
  1. The demand is calculated 12 hours before.
  2. The local management takes an average of one hour to process a demand.
  3. Renewable resources take two hours in order to supply the power, pollution resources take one hour and high-risk resource take five hours.
  4. The supply for adjacent areas takes one hour over the above times.
  5. The central management takes one hour to process a demand, as the local management.

Figure 5 shows a particular RT-UML sequence scene of this system. This scene captures the situation for attending the demand of a specific area. This request is performed by a "demand" message, which indicates the area in which the source of the demand has been generated and the total power amount that will be necessary in 12 hours. The destination of this request is the local management, which has one hour to decide if the "local resources" have enough power to attend the demand (depicted in Fig. 5 with the labeled frame "choice1") or to send the demand to the "central management" (depicted in Fig. 5 with the labeled frame "choice2"). This decision is performed by dealing with two factors: environmental factors and resources availability, as mentioned previously in the restrictions. If the decision is to perform "choice1", then the "local management" sends "supply" messages to the corresponding local resources, which indicate the type and amount of power that must be generated in order to attend this demand. However, if the decision is "choice2", then the "local management" informs the "central management" by sending a "demand" message with the requested amount. The "central management" has one hour to deliver this request to another "local management", in which there are available resources to attend the demand or part of this demand. Note that this information is available due to the "local management" informing the "central management" with "supply" messages, which contain information about the availability of resources and

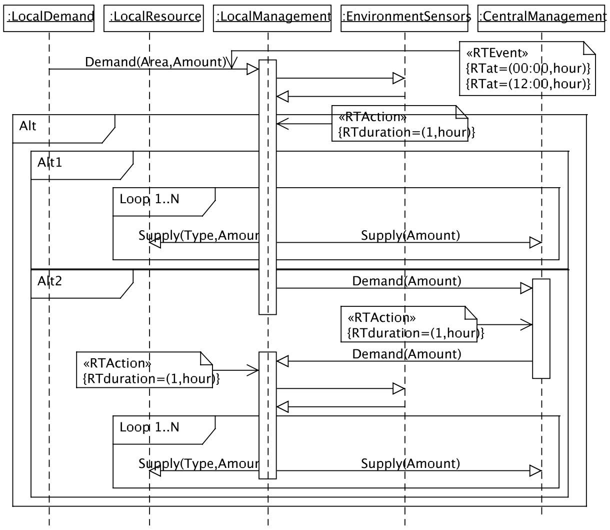


Fig. 5. Sequence diagram for the PMS.

the amount of power that is going to be generated. Figure 6 shows an activity diagram for the "local management" described in this sequence scene.

Figure 7 shows a part of the WS-BPEL specification for a "local management" process, which has been obtained by applying the transformation rules from the previous section. This piece of the specification corresponds to the situation in which the "local management" receives a "demand" request from the "local demand". For instance, in this specification, we can see how the RT-Event "demand" is transformed into a wait-until structure in which a receive message is specified and the decision performed between "choice1" and "choice2" is transformed into an **if** control structure. For a better readability, we have replaced some Xpath expressions, which are too complex and large, with natural language sentences.

6 Conclusions and Future Work

In this paper we have presented a translation of RT-UML Sequence and Activity diagrams into WS-BPEL documents. Sequence diagrams precisely describe the collaboration and message flow between the parties involved in a choreography, and also capture the time windows in which the conversations can take place. On the other hand, activity diagrams describe in detail the way in which the different actions are done, by means of sequence, choice, parallelism and time outs.

The descriptions of Web Services by WS-BPEL documents, written in XML format, can be very difficult to understand for non-experts, so we use RT-UML in order to design these Real Time Web Services.

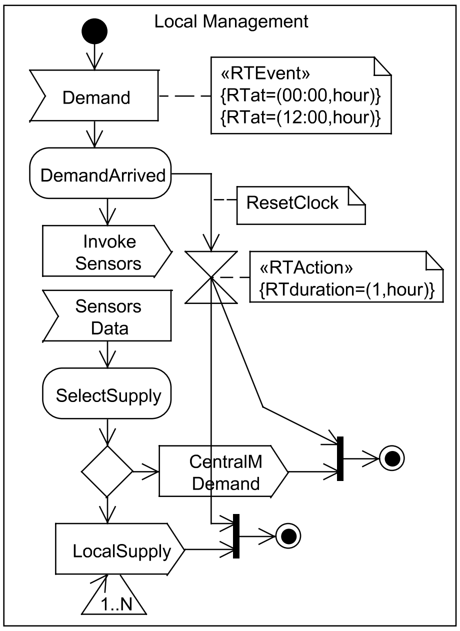


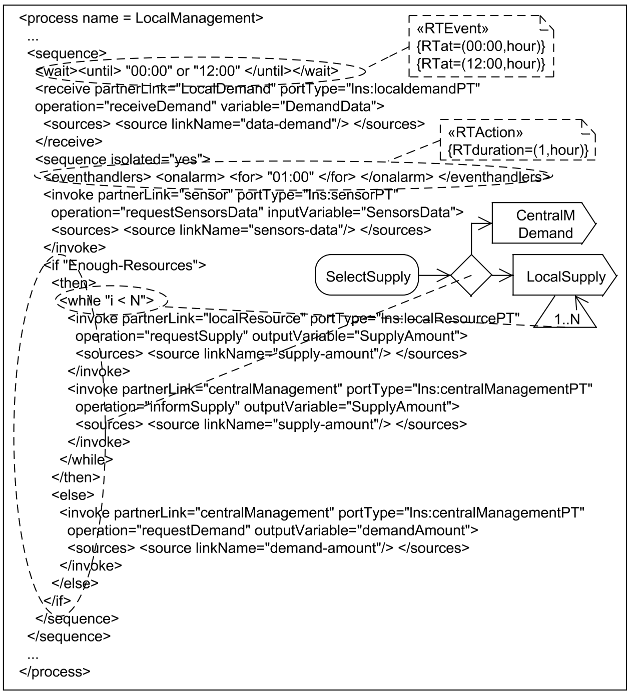
Fig. 6. Activity diagram for a "Local Management".

Our methodology can exploit some previous works, in which both WS-CDL and WS-BPEL descriptions are translated into timed automata, thus allowing us to simulate and analyze the system behavior, for instance by using the UPPAAL tool.

Observe that the automatic generation of WS-BPEL documents, which are very close to an implementation, provides us with a correct design, at least with respect to the RT-UML Diagrams of reference.

The translation presented is therefore a powerful tool for obtaining correct Web Services. We have established the equivalence between the different elements of the RT-UML sequence and activity diagrams with the elements of the WS-BPEL specifications, as a design method, and the translation of these WS-BPEL documents to Timed Automata, in order to perform Model Checking on them. Furthermore, these translations have been illustrated with an example.

Our future work addresses the issue of developing a tool capturing all these capabilities: RT-UML modeling, translation to WS-BPEL and WS-CDL, as well as the generation of timed automata XML files. We have a preliminary version of the tool that can be found in the URL http://www.info-ab.uclm.es/fmc/tools/WS-CDLUppaal.rar.



```
<process name = LocalManagement>
  ...
  <sequence>
   <wait><until> "00:00" or "12:00" </until></wait>
   <receive partnerLink="LocalDemand" portType="lns:localdemandPT"
   operation="receiveDemand" variable="DemandData">
     <sources> <source linkName="data-demand"/> </sources>
   </receive>
   <sequence isolated="yes">
    <eventhandlers> <onalarm> <for> "01:00" </for> </onalarm> </eventhandlers>
    <invoke partnerLink="sensor" portType="lns:sensorPT"
      operation="requestSensorsData" inputVariable="SensorsData">
      <sources> <source linkName="sensors-data"/> </sources>
    </invoke>
    <if "Enough-Resources">
     <then>
      <while "i < N">
        <invoke partnerLink="localResource" portType="lns:localResourcePT"
          operation="requestSupply" outputVariable="SupplyAmount">
          <sources> <source linkName="supply-amount"/> </sources>
        </invoke>
        <invoke partnerLink="centralManagement" portType="lns:centralManagementPT"
          operation="informSupply" outputVariable="SupplyAmount">
          <sources> <source linkName="supply-amount"/> </sources>
        </invoke>
      </while>
     </then>
     <else>
      <invoke partnerLink="centralManagement" portType="lns:centralManagementPT"
        operation="requestDemand" outputVariable="demandAmount">
        <sources> <source linkName="demand-amount"/> </sources>
      </invoke>
     </else>
    </if>
   </sequence>
  </sequence>
  ...
</process>
```

Fig. 7. WS-BPEL process specification of a Local Management party.

References

1. Jean Paoli, Eve Maler, and et. al. Tim Bray. Extensible markup language (xml) 1.0 (third edition), 2004. http://www.w3.org/TR/2004/REC-xml-20040204.
2. Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. Uddi version 3.0.2, 2004. http://uddi.org/pubs/uddi_v3.htm.
3. Marc Hadley, Noah Mendelsohn, and et. al Jean-Jacques Moreau. Soap version 1.2 part 1: Messaging framework, 2003. http://www.w3.org/TR/soap12-part1.
4. Sanjiva Weerawarana, Roberto Chinnici, and et. al. Martin Gudgin. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2004. http://www.w3.org/TR/2004/WD-wsdl20.
5. Assaf Arkin, Sid Askary, and et. al. Ben Bloch. Web Services Business Process Execution Language Version 2.0, December 2004. http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm.
6. Nickolas Kavantzas et al. Web Service Choreography Description Language (WSCDL) 1.0. http://www.w3.org/TR/ws-cdl-10/.
7. G. Diaz, M. E. Cambronero, M.LL. Tobrarra, V. Valero, and F. Cuartero. Analysis and Verification of Time Requirements Applied to the Web Services Composition. In *Proceedings of WS-FM, Viena*, Lecture Notes in Computer Science, pages 178–192. Springer, September 2006.

8. G. Diaz, M. E. Cambronero, J. J. Pardo, V. Valero, and F. Cuartero. Automatic Generation of Correct Web Services Choreographies and Orchestrations with Model Checking Techniques. In *Proceedings of International Conference on Internet and Web Applications and Services ICIW'06*. IEEE Press.
9. G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Verification of Web Services with Timed Automata. In *Proceedings of First International Workshop on Automated Specification and Verification of Web Sites*, Electronic Notes in Theoretical Computer Science.
10. G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Automatic Translation of WS-CDL Choreographies to Timed Automata. In *Proceedings of WS-FM, Versalles*, Lecture Notes in Computer Science, pages 230–242. Springer, September 2005.
11. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
12. Keith Mantell. From UML to BPEL, September 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel.
13. David Skogan, Roy Grønmo, and Ida Solheim. Web service composition in uml. In *EDOC*, pages 47–57, 2004.
14. OMG. *UML 2.0 Superstructure proposal v.2.0.*, January 2003.
15. UML Profile for Schedulability, Performance, and Time Specification, Version 1.1. In http://www.omg.org/docs/smsc/04-12-05.pdf.
16. David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
17. James Clark. XSL Transformations (XSLT) Version 1.0. Technical Report REC-xml-19980210, W3C, 1998. `http://www.w3.org/TR/xslt`.
18. Artisan tool 6.0. *Artisan Real Time Studio*, 2001.

Applying Model Intelligence Frameworks for Deployment Problem in Real-Time and Embedded Systems

Andrey Nechypurenko,
Egon Wuchner
Siemens AG,
Corporate Technology (SE 2)
Otto-Hahn-Ring 6
81739 Munich, Germany
{andrey.nechypurenko,
egon.wuchner}@siemens.com

Jules White,
Douglas C. Schmidt
Vanderbilt University,
Department of Electrical Engineering and
Computer Science
Box 1679 Station B
Nashville, TN, 37235, USA
{jules, schmidt}@dre.vanderbilt.edu

ABSTRACT

There are many application domains, such as distributed real-time and embedded (DRE) systems, where the domain constraints are so restrictive and the solution spaces so large that it is infeasible for modelers to produce correct solution manually using a conventional graphical model-based approach. In DRE systems the available resources, such as memory, CPU, and bandwidth, must be managed carefully to ensure a certain level of quality of service. This paper provides three contributions to simplify modeling of complex application domains: (1) we present our approach of combining model intelligence and domain-specific solvers with model-driven engineering (MDE) environments, (2) we show techniques for automatically guiding modelers to correct solutions and how to support the specification of large and complex systems using intelligent mechanisms to complete partially specified models, and (3) we present the results of applying an MDE tool that maps software components to Electronic Control Units (ECUs) using the AUTOSAR automotive modeling and middleware standard.

1. INTRODUCTION

Graphical modeling languages, such as UML, can help to visualise certain aspects of the system and automate particular development steps via code-generation. Model-driven engineering (MDE) tools and domain-specific modeling languages (DSMLs) [7] are graphical modeling technologies that combine high-level visual abstractions that are specific to a domain with constraint checking and code-generation to simplify the development of certain types of systems. In many application domains, however, the domain constraints are so restrictive and the solution spaces so large that it is infeasible for modelers to produce correct solutions manually. In these domains, MDE tools that simply provide solution correctness checking via constraints provide few benefits over conventional approaches that use third-generation languages.

Regardless of the modeling language and notation used, the inherent complexity in many application domains is the combinatorial nature of the constraints, and not the code construction per se. For example, specifying the deployment of software components to hardware units in a car in the face of configuration and resource constraints can easily generate solution spaces with millions or more possible deployments and few correct ones, even when only scores of model entities are present. For these combinatorially complex modeling problems, it is impractical, if not impossible, to create a complete and valid model manually. Even connecting hundreds of components to scores of nodes by pointing and clicking via a GUI is tedious and error-prone. As the number of modeling elements increases into the thousands, manual approaches become infeasible.

To address the challenges of modeling combinatorially complex domains, therefore, we need techniques to reduce the cost of integrating a graphical modeling environment with *Model Intelligence Guides (MIGs)*, which are automated MDE tools that help guide users from partially specified models, such as a model that specifies components and the nodes they need to be deployed to but not how they are deployed, to complete and correct ones, such as a model that not only specifies the components to be deployed but what node hosts each one. This paper describes techniques for creating and maintaining a *Domain Intelligence Generator* (DIG), which is an MDE that helps modelers solve combinatorially challenging modeling problems, such as resource assignment, configuration matching, and path finding.

The rest of the paper is organised as follows: Section 2 discusses challenges of creating deployment models in the context of the AUTOSAR[3] middleware and modeling standard, which we use as a motivating example; Section 3 describes key concepts used to create and customize MIGs; Section 4 shows the results of applying MIGs to AUTOSAR component deployments; and Section 5 presents concluding remarks and outlines future work.

MODELS06WS6 ’06 Genova, Italy

2. MOTIVATING EXAMPLE

AUTOSAR is a new standard for automotive middleware and software development modeling [3]. The goal of AUTOSAR is to standardize solutions to many problems that arise when developing large-scale, distributed real-time and embedded (DRE) systems for the automotive domain. For instance, concert efforts is required to relocate components between Electronic Control Units (ECUs), *i.e.*, computers and micro-controllers running software components within a car. Key complexities of relocation include: (1) components often have a many constraints that need to be met by the target ECU and (2) there are many possible deployments of components to ECUs in a car and it is hard to find the optimal one.

For example, it is hard to manually find a set of interconnected nodes able to run a group of components that communicate via a bus. Modelers must determine whether the available communication channels between the target ECUs meet the bandwidth, latency, and framing constraints of the components that communicate through them. In the automotive domain—as with other embedded systems domains— it is also important to reduce the overall cost of the solution, which necessitates optimizations, such as finding deployments that use as few ECUs as possible or minimize bandwidth to allow cheaper buses. It is infeasible to find these solutions manually for a production systems.

To illustrate the practical benefits of generating and integrating MIGs with a DSML, we describe an MDE tool we developed to solve AUTOSAR constraints for validly deploying software components to ECUs. There are two primary architectural views in AUTOSAR systems:

- The *logical collaboration structure* that specifies which components that should communicate with each other via which interfaces, and
- The *physical deployment structure* that captures the capabilities of each ECU, their interconnecting buses, and their available resources.

Historically, AUTOSAR developers have manually specified the mapping from components in the logical view to ECUs in the physical view via MDE deployment tools, as shown in Figure 1. This approach worked relatively when when there were a small number of components and ECU. Modern cars, however, can be equipped with 80 or more ECUs and several hundred or more software components. Simply drawing arrows from 160 components to 80 ECUs is tedious. Moreover, many requirements constrain which ECUs that can host certain components, including the amount of memory required to run, CPU power, programming language, operating system type and version, etc. These constraints must be considered carefully when deciding where to deploy a particular component. The problem is further exacerbated when developers consider the physical communication paths and aspects, such as available bandwidth in conjunction with periodical real-time messaging.

The remainder of this paper how the AUTOSAR MDE tool we developed helps automate the mapping of software components to ECUs in AUTOSAR models without violating the known constraints. The following sections describe our approach and show how MIGs can significantly reduce the complexity of creating AUTOSAR deployment models.

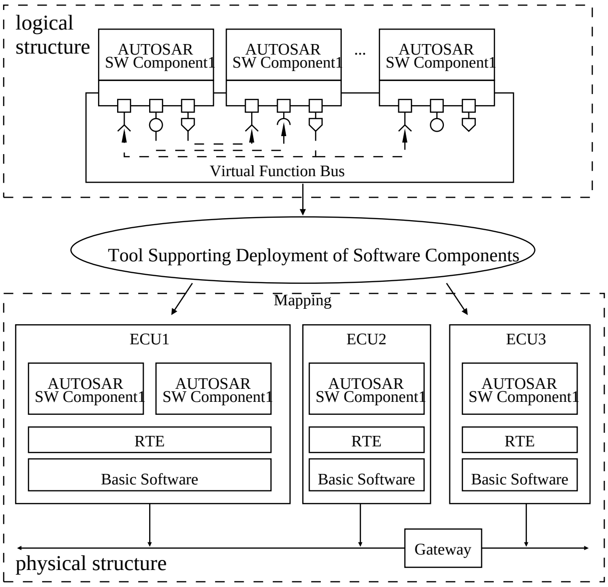


Figure 1: Mapping from the logical collaboration to the physical deployment structure

3. DOMAIN-SPECIFIC MODEL INTELLIGENCE

Based on the challenges related to the AUTOSAR example presented in Section 2, the goals of our work on MIGs are to (1) specify an approach for guiding modelers from partially specified models to complete and coorrect ones and (2) automate the completion of partially specified models using information extracted from domain constraints.

In previous work [9, 8], we showed how MDE tools and DSMLs can improve the modeling experience and bridge the gap between the problem and solution domain by introducing domain-specific abstractions. At the heart of these efforts is the *Generic Eclipse Modeling System* (GEMS), which provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the DSML. Given a metamodel, GEMS automatically generates a graphical editor that enforces the grammar specified in the DSML. GEMS provides convenient infrastructure (such as built-in support for the Visitor pattern[5]) to simplify model traversal and code generation. We used GEMS as the basis for our MIGs AUTOSAR deployment modeling tool and our work on domain-specific model intelligence.

3.1 Domain Constraints as the Basis for Automatic Suggestions

A key research challenge was determining how to specify the set of model constraints so they could be used by MIGs *not only to check the correctness of the model, but also to guide users through a series of model modifications to bring it to a state that satisfies the domain constraints.* We considered various approaches for constraint specification language, including Java, the Object Constraint Language (OCL), and Prolog. To evaluate the pros and cons of each approach, we implemented our AUTOSAR deployment constraints in each of the three languages.

As a result of this evaluation, we selected Prolog since it provided both constraint checking and model suggestions. In particular, Prolog can return the set of possible facts from a knowledge base that indicate why a rule evaluated to "true." The declarative nature of Prolog significantly reduced the number of lines of code written to transform an instance of a DSML into a knowledge base and to create constraints (its roughly comparable to OCL for writing constraints). Moreover, Prolog enables MIGs to derive sequences of modeling actions that converts the model from an incomplete or invalid state to a valid one. As shown in Section 1, this capability is crucial for domains, such as deployment in complex DRE systems, where manual model specification is infeasible or extremely tedious and error-prone.

The remainder of this section describes how Domain Intelligence Generation (DIG) uses Prolog and GEMS to support the creation of customizable and extensible domain-specific constraint solver and optimization frameworks for MIGs. Our research focuses on providing modeling guidance and automatic model completion, as described below.

3.2 Modeling Guidance on-the-fly

To provide domain-specific model intelligence, an MDE tool must capture the current state of a model and reason about how to assist and guide modelers. To support this functionality, MIGs use a Prolog knowledge base format that can be parameterized by a metamodel to create a domain-specific knowledge base. GEMS metamodels represent a set of model entities and the role-based relationships between them. For each model, DIG populates a Prolog knowledge base using these metamodel-specified entities and roles. For each entity, DIG generates a unique id and a predicate statement specifying the type associated with it.

In the context of our AUTOSAR example, a model is transformed into the predicate statement *component(id)*, where id is the unique id for the component. For each instance of a role-based relationship in the model, a predicate statement is generated that takes the id of the entity it is relating and the value it is relating it to. For example, if a component with id 23 has a *TargetHost* relationship to a node with id 25 the predicate statement *targethost(23,25)* is generated. This predicate statement specifies that the entity with id 25 is a *TargetHost* of the entity with id 23. Each knowledge base generated by DIG provides a domain-specific set of predicate statements.

The domain-specific interface to the knowledge base provides several advantages over a generic format, such as the format used by a general-purpose constraint solver like for exampl CLIPS. First, the knowledge base maintains the domain-specific notations from the DSML, making the format more intuitive and readable to domain experts. Second, maintaining the domain-specific notations allows the specification of constraints using domain notations, thereby enabling developers to understand how requirements map to constraints. Third, in experiments that we conducted, writing constraints using the domain-specific predicates produced rules that had fewer levels of indirection and thus outperformed rules written using a generic format. In general, the size of the performance advantage depended on the generality of the knowledge base format. To access properties of the model entities, the predicate syntax presents the most specific knowledge base format. Given an entity id and role name, the value can be accessed with the statement *role(id, Value)*, which has exactly zero or one facts that match it.

Based on this domain-specific knowledge base, modelers can specify user-defined constraints in form of Prolog rules for each type of metamodel relationship. These constraints semantically enrich the model to indicate the requirements of a correct model. They are also used to automatically deduce the sets of valid model changes to create a correct model.

For example, consider the following constraint to check if a node (ECU) is a valid host of a component: *is_a_valid_component_targethost(Comp, Nodes)*. It can be used to both check a Component/Node combination (*e.g.*, *is_a_valid_component_targethost(23,[25])*.) and to find valid *Nodes* that can play the *TargetHost* role for a particular component (*e.g.*, *is_a_valid_component_targethost(23, Nodes)*.). The latter example uses Prolog's ability to deduce the correct solution, *i.e.*, the *Nodes* variable will be assigned with the list of all constraint-valid nodes for the *TargetHost* role of the specified component. This example illustrates how constraints can be used to check *and* to generate the solution, if one exists.

Figure 2 shows how dynamic suggestions from Prolog are presented to modelers. The upper part of the figure shows the fragment of the metamodel that describes the *Deployment* relationship between *Component* and *Node* model entities. The lower part of the picture shows how the generated editor displays the corresponding entity instances. This screenshot was made at the moment a modeler had begun dragging a connection begining from the "ABS" compo-

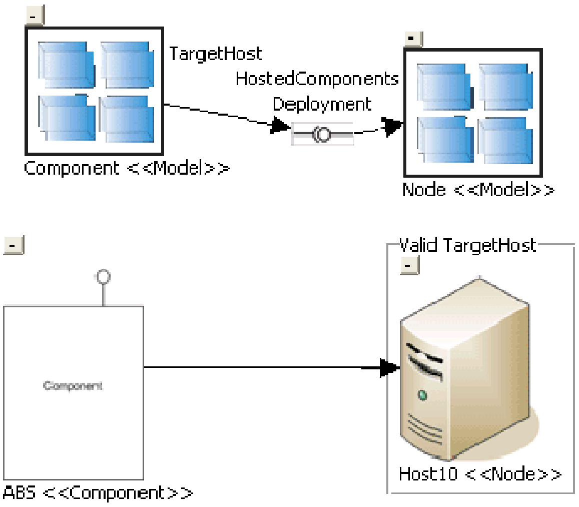


Figure 2: Highlighting valid target host

nent. The rectangle around "Host10" labelled "Valid TargetHost" is drawn automatically as a result of triggering the corresponding solver rule and receiving a valid solution as feeback. GEMS also can also trigger arbitrary Prolog rules from the modeling tool and incorporate their results back into a model. This mechanism can be used to solve for complete component to ECU deployments and automatically add deployment relationships based on a (partially) complete model.

To enable modeling assistance, different subsystems must collaborate within the modeling environment. It is the responsibility of the modeler (or MDE tool creator) to provide the set of constraints and supply solvers for new constraint types. The GEMS metamodel editor updates the knowledge base and incorporates the new rules into the generated MIG. User-defined solver(s) can be based on existing Prolog algorithms, the reusable rules generated by GEMS, or a hybrid of both. Solvers form the core of the basic MIG generated by GEMS. Below we describe the solver we developed for completing partially specified models in our AUTOSAR deployment tool.

3.3 Model Completion Solvers

Using a global deployment (completion) solver, it is possible to ask for the completion of partially specified models constrained by user-defined rules. For example, in the AUTOSAR modeling tool, the user can specify the components, their requirements, the nodes (ECUs), and their resources and ask the tool to find a valid deployment of components to nodes. After deploying the most critical components to some nodes by using MIGs step-wise guidance, modelers can trigger a MIG global deployment solver to complete the deployment. This solver attempts to calculate an allocation of components to nodes that observes the deployment constraints and update the connections between components and nodes accordingly. This global solver can aim for an optimal deployment structure by using constraint-based Prolog programs or it could integrate some domain-specific heuristics, such as attempting to find a placement for the components that use the most resources first.

In some cases, however, the modeled constraints cannot be satisfied by the available resources. For example, in a large AUTOSAR model, a valid bin-packing of the CPU requirements for the components into EPUs may not exist. In these cases the complexity of the rules and entity relationships could make it extremely hard to deduce *why* there is no solution and *how* to change the model to overcome the problem. For such situations, we developed a solver that can identify failing constraints and provide suggestions on how to change the model to make the deployment possible.

4. CASE STUDY: SOLVING AUTOSAR DEPLOYMENT PROBLEM

To validate our DIG MDE tool, we created a DSML for modeling AUTOSAR deployment problems. This DSML enables developers to specify partial solutions as sets of components, requirements, nodes (ECUs), and resources. A further requirement was that the MIGs should produce both valid assignments for a single component's *TargetHost* role and global assignments for the *TargetHost* role of all components. In the automotive domain certain software components often cannot be moved between ECUs from one model car to the next due to manufacturing costs, quality assurance, or other safety concerns. In these situations, developers must fix the *TargetHost* role of certain components and allow MIGs to solve for valid assignments of the remaining unassigned component *TargetHost* roles.

For the first step, we created a deployment DSML metamodel that allows users to model components with arbitrary configuration and resource requirements and nodes (ECUs) with arbitrary sets of provided resources. Each component configuration requirement is specified as an assertion on the value of a resource of the assigned *TargetHost*. For example, *OSVersion* $>$ *3.2* would be a valid configuration constraint. Resource constraints were created by specifying a resource name and the amount of that resource consumed by the component. Each Node could only have as many components deployed to it as its resources could support. Typical resource requirements were the RAM usage and CPU usage.

Each host can provide an arbitrary number of resources. Constraints comparisons on resources were specified using the $<$, $>$, -, and $=$ relational operators to denote that the value of the resource with the same name and type (*e.g.*, OS version) must be less, greater, or equal to the value specified in requirement. The "-" relationship indicates a summation constraint, *i.e.*, the total value of the demands on a resource by the components deployed to the providing node must not exceed the amount present on the node. After defining the metamodel and generating the graphical editor for the deployment DSML using GEMS, we added a set of Prolog constraints to enforce the configuration and resource constraint semantics of our models.

4.1 Defining Constraints and Solvers

Our constraint rules specified that for each child requirement element of a component, a corresponding resource child of the TargetHost must satisfy the requirement. Our complete configuration constraint rules are as following.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% specifying validness of a requirement-resource
% pair
requirement_resource_valid_pair(Req, Res) :-
    (requirement_spec(Req, Name, '>'),!
     ;
     requirement_spec(Req, Name, '<'),!
     ;
     requirement_spec(Req, Name, '=')
    ),
    resource_spec(Res, Name, '=').

requirement_to_resource(Req, Host, Res) :-
    requirement(Req),
    resource_to_node(Res, Host),
    requirement_resource_valid_pair(Req, Res).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% configuration requirement resource solver
comparevalue(V1,V2,'>') :- V1 > V2.
comparevalue(V1,V2,'<') :- V1 < V2.
comparevalue(V1,V2,'=') :- V1 == V2.
```

```
requirement_resource_constraint(Req, Res) :-
    requirement(Req),
    self_type(Req, Type),
    (Type = '<' ; Type = '>' ; Type = '='),
    !,
    resource(Res),
    self_value(Res, ResValue),
    self_value(Req, ReqValue),
    comparevalue(ResValue, ReqValue, Type).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% local role base component targethost
% relationship solver
is_a_valid_component_targethost(Owner, Value) :-
( self_targethost(Owner, [Value]), ! %deployed
  ;
  (is_a(Value,node),
   self_requires(Owner, Requirements),
   forall( member(Req,Requirements)
     ,
     (requirement_to_resource(Req, Value, Res),
     requirement_resource_constraint(Req, Res))
) ) ).
```

These lines of code are the *entire* solution, providing not only configuration constraint checking for an arbitrary set of requirements and resources but also enabling domain-specific GEMS editors to provide valid suggestions for deploying a component. Moreover, this solution was intended as a proof-of-concept to validate the approach and thus could be implemented with even fewer lines of code. The rest of the required predicates to implement the solver were generated by GEMS.

In our experiments with global solvers, Prolog solved a valid global deployment of 900 components to 300 nodes in approximately 0.08 seconds. This solution met all configuration constraints.

The rules required for solving for valid assignments using resource constraints were significantly more complicated since resource constraints are a form of bin-packing (an NP-Hard problem). We were able to devise heuristic rules in Prolog, however, that could solve a 160 component and 80 ECU model deployment in approximately 1.5 seconds and an entire 300 component and 80 ECU deployment, a typical AUTOSAR sized problem, in about 3.5 seconds. These solution times are directly tied to the difficulty of the problem instance. For certain instances, times could be much higher, which would make the suggestive solver from Section 3 discussed in the previous section applicable. In cases where the solver ran too long, the suggestive solver could be used to suggest ways of expanding the underlying resources and making the problem more tractable.

5. RELATED WORK

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [2, 6, 4]. These tools provide mechanisms for describing domain-constraints, a set of knowledge, and finding solutions to the constraints. These tools, however, are not designed to generated domain-specific solvers based on a metamodel. These tools also do not support the generation of a DSML graphical environment and integrated graphical suggestions. In contrast, our domain-specific model intelligence, based on GEMS, is automatically integrated with any DSML tool generated from a GEMS metamodel.

Decision support systems are similar to the domain-specific model intelligence approach proposed in this paper. In [1], Achour and all propose a modeling tool based on the Unified Medical Language System (UMLS), to create knowledge bases for diagnosing and treating diseases. Both their UMLS approach and our approach attempt to glean domain knowledge and constraints from an expert and simplify users abilities to find the correct solution to a partially specified problem. Their approach, however, differs significantly from our approach in several ways. First, our approach is designed to facilitate the creation of decision support systems for any domain-specific modeling language. In particular, MIGs are not limited solely to decision tree type guidance but also complex analysis and optimizations specified by users. Second, MIGs are automatically generated from a metamodel and integrated with a graphical modeling tool via GEMS, which supports the creation of graphical modeling tools with integrated modeling decision support for arbitrary domains.

6. CONCLUDING REMARKS

The work presented in this paper addresses scalability problems of conventional manual modeling approaches. These scalability issues are particularly problematic for domains that have large solutions spaces and few correct solutions. In such domains, it is often infeasible to create correct models manually, so constraint solvers are therefore needed.

Turning a DSML instance into a format that can be used by a constraint solver is a time-consuming task. Our DIG MDE tool generates a domain-specific constraint solver that leverages a semantically rich knowledge base in Prolog format. It also allows users to specify constraints in declarative format that can be used to derive modeling suggestions.

In future work, we plan to continue our development of templatized solver-frameworks for modeling tools and incorporate new types of constraint solvers into the framework. We plan to investigate the use of both automatic control and monitoring of running systems using domain-specific model intelligence and human-assisted monitoring and control. Finally, we intend to extend our infrastructure to allow other types of constraint solving platforms, such as bin-packing solvers written in C, to be integrated into a GEMS-based modeling environment.

GEMS and the MIGs generation framework is an open-source project available from:
http://www.sf.net/projects/gems.

7. REFERENCES

[1] S. L. Achour, M. Dojat, C. Rieux, P. Bierling, and E. Lepage. A umls-based knowledge acquisition tool for rule-based clinical decision support system development. *Journal of the American Medical Information Association*, 8(4):351–360, July 2001.

[2] J. Cohen. Constraint logic programming languages.

Commun. ACM, 33(7):52–68, 1990.

[3] H. H. et al. Autosar current results and preparations for exploitation. In *7th EUROFORUM conference*, may 2006.

[4] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, November 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[6] G. Smolka. The oz programming model. In *JELIA '96: Proceedings of the European Workshop on Logics in Artificial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[7] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[8] J. White and D. C. Schmidt. Simplifying the development of product-line customization tools via mdd. In *Workshop: MDD for Software Product Lines, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, October 2005.

[9] J. White and D. C. Schmidt. Reducing enterprise product line architecture deployment costs via model-driven deployment and configuration testing. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2006.

An experience in modeling real-time systems with SysML

Pietro Colombo Vieri Del Bianco Luigi Lavazza Alberto Coen-Porisini

Dipartimento di Informatica e Comunicazione
Università dell'Insubria
Via Mazzini, 5
21100 Varese, Italia

{pietro.colombo, vieri.delbianco, luigi.lavazza, alberto.coenporisini}@uninsubria.it

ABSTRACT

Model-based development is particularly attractive and promising in the area of real-time and embedded systems, since the complexity of such systems can greatly benefit from the availability of models that abstract away details, allowing developers to focus on the most relevant and challenging features of the systems.

Recently, SysML has been adopted by the OMG as a modeling language for Systems Engineering. SysML is a UML profile that represents a subset of UML 2 with extensions. These characteristics make SysML particularly interesting, since it remains largely compatible with UML while overcoming its limits concerning the possibility to model non software elements of systems, and developing a systemic view in the models.

Since SysML has been released quite recently, it is not yet clear how effectively SysML can be used to model the different relevant aspects of real-time and embedded systems, which have some very specific requirements.

This paper reports a modeling activity that was carried out in order to test the effectiveness of SysML in modeling real-time and embedded systems. In particular, SysML was used to model one of the best known benchmarks for real-time development languages: the crossing gate problem. By means of this experience the authors were able to perform a first assessment of the strengths and the weaknesses of SysML as a real-time modeling language.

Keywords

Real-time software, Model-based development, SysML.

1 INTRODUCTION

The usage of models in software development is continuously increasing. This practice makes it possible to concentrate on the essential features of the systems without having to deal with the intricacies of actual code. Moreover, models are usually platform independent, thus keeping application development separate from the underlying platform technology in the early stages of the lifecycle, and easing porting. Whenever sufficiently expressive modeling languages are used, models can also be processed for various purposes, such as to validate or/and verify properties, to generate code, to generate test cases, etc.

Recently, SysML has been proposed by the OMG as a modeling language for Systems Engineering [7]. It is a UML profile based on a subset of UML 2 [10][11] with extensions. It introduces new features that are expected to better support the specification, analysis, design, verification, and validation of *systems* that include hardware, software, data, personnel, procedures, and facilities.

Since real-time and embedded systems are not comprised uniquely of software, but tend to involve hardware, devices, and often people, SysML is a natural candidate to support the development of this type of systems. Nevertheless, since SysML was released quite recently, there is still little evidence of its suitability to support effectively the development of real-time embedded systems, which have some very specific requirements.

The goal of this paper is to test the effectiveness of SysML in modeling real-time and embedded systems. In the past, the authors were involved in the definition of techniques for model based development of real-time and embedded software using UML and extensions of UML [14][15][16][17][18][19]. In that activity, it was particularly enlightening the usage of UML to model one of the best known benchmarks for real-time development languages: the generalized crossing gate (GRC) problem [2]. Here the experience is repeated with SysML: by modeling the GRC with SysML we were able to perform a first assessment of the strengths and the weaknesses of the language.

The paper is organized as follows: Section 2 introduces the GRC problem, Section 3 illustrates the corresponding SysML model. Section 4 discusses the effectiveness of SysML, and identifies some qualities and limitations. Section 5 draws some conclusions.

Throughout the paper we assume that the reader has a minimum of familiarity with SysML.

2 THE GRC PROBLEM

The Generalized Railroad Crossing problem was proposed by Heitmeyer [2] as a general benchmark for the development of real-time systems. Since then it has been used extensively to test several formalisms, methodologies and tools aimed at easing the development of real-time systems [2][3]. The problem is quite simple, yet it exposes time constraints we usually find in hard real-time systems.

2.1 The definition

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest *R* (see Figure 1). Trains travel over *R* on *K* tracks in both directions. Trains may proceed at different speeds, and can even pass each other. Only one train per track is allowed to be in *R* at any moment. Sensors indicate when each train enters and exits regions *R* and *I*. A gate function *g* from the real-time domain to the real interval [0,90] describes the state of the gate according to the inclination of the bar, *g(t)=0* indicating that the bar is down (gate closed) and *g(t)=90* indicating that the bar is up (gate open). A sequence of “occupancy intervals” is also defined, where each occupancy interval is the maximal time interval during which one or more trains are in *I*.

The problem is to develop a system to operate the crossing gate that satisfies the following two properties:

- *Safety Property*: The gate is closed during all occupancy intervals.
- *Utility Property:* The gate is open whenever this is possible without violating the safety property, and according to the features of the gate. For instance, when the last train in the crossing leaves and no train is approaching, the gate must open. The utility property is required, since a permanently closed gate satisfies the safety property.

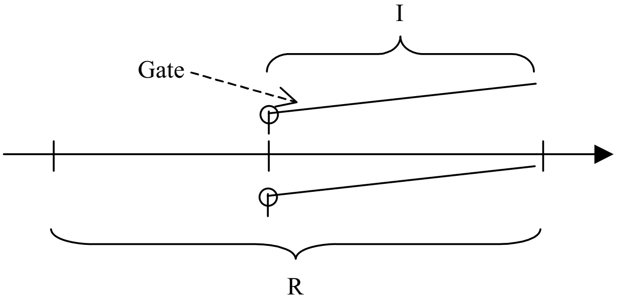


Figure 1. The railroad crossing regions.

Notice that Figure 1 shows trains going in only one direction. We adopt this simplification, since it has been shown that the solution of this simplified problem can be trivially extended to the general case.

2.2 Towards the solution of the GRC problem

Let us introduce the relevant points in the interest region (see Figure 2):

- point *RI* indicates the position of the entrance sensor;
- point *RO* indicates the position of the exit sensor;

- point *II* indicates the position of the sensor which detects trains entering region *I*.

RI-RO defines zone *R* and *II-RO* defines zone *I*.

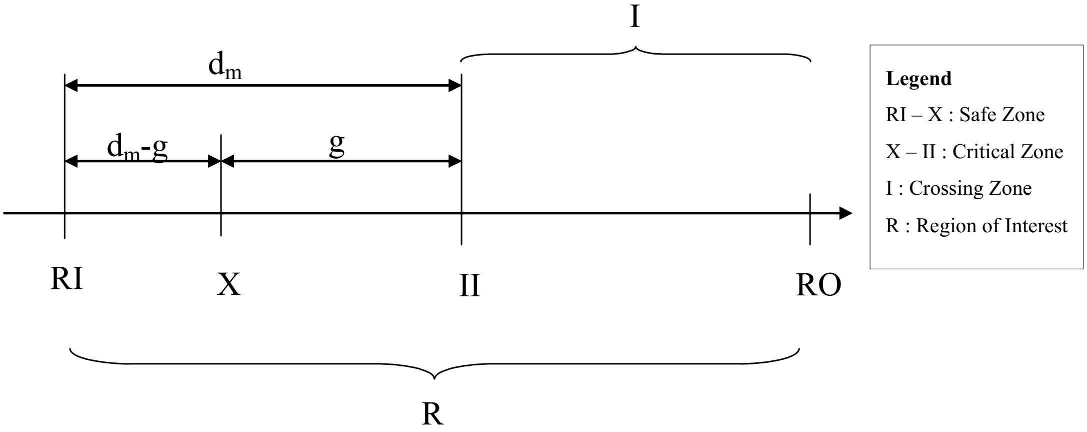


Figure 2. Annotated GRC.

A set of temporal constants describe maximum and minimum times for crossing the various zones, as well as the gate opening and closing times:

- *dm* and *dM*: minimum and maximum time taken by a train to cross *RI-II* zone;
- *hm* and *hM*: minimum and maximum time taken by a train to cross zone *I* (i.e., *II-RO* zone);
- *g* is the time taken by the bars of the gate for moving from the completely open to completely closed position (or vice versa).

Point *X* is thus defined as follows: when a train enters zone *X-II* it is time to start closing the gate, so that bars will be completely lowered when the train arrives at *II* (i.e., when the train enters the crossing zone *I*, or *II-RO*). We name *RI-X* and *X-RO* zones “Safe zone” and “Critical zone”, respectively. The exact position of *X* depends on the speed of each train, which is not known precisely. Thus the system cannot determine the right moment when a given train is at point *X*. However, it is clear that if we make the system safe for the fastest train, it will be safe also for other trains. In order to have the gate closed when the fastest trains arrive at *II*, we must begin to close the gate dm-g seconds after the train entered region *R*. In this way when the fastest trains arrive at *II* the bars will be down and the crossing will be safe. Of course, the system will be safe for the slower trains as well.

In order to assure the safety property, the bars can be raised only when both the crossing zone and the critical zone are empty. Similarly, in order to assure the utility property, the system must start opening the gate as soon as the critical zone and the crossing zone become empty.

3 MODELING THE GRC WITH SysML

The System Modeling Language (SysML) is a visual modeling language that provides a graphical notation with (informal) semantics and different types of diagrams, which can be used to describe both the behavioral and the structural aspects of a system [9]. The variety of diagrams and a very rich notation allow designers to show in a quite intuitive way the relationships among the different elements involved in a model. Most of the diagrams are inherited from UML and are adopted without any change. Other diagrams are adapted so that they are compatible with the new constraints defined by SysML. Moreover, completely new diagrams are defined to extend the language in order to fully support system modeling.

SysML does not provide a specific modeling methodology. The language is said methodology independent, and thus the system engineer is completely free to adopt the approach he/she likes best. In the following sections we will see how SysML can be used to support the modeling activity of the GRC problem. The methodology adopted follows a classical iterative approach. Each step contributes to the growth of the whole model but sometimes it is necessary to go back and modify some element in order to assure the consistency of the system. We will start describing the requirements of the system using a Requirements diagram. Then we will define the structural aspects with Block Definition diagrams and Internal Block diagrams. Behavioral aspects will be defined using Activity diagrams and State Machine diagrams, while constraints on the model elements will be introduced using Parametric diagrams. The usage of cross cutting constructs, like allocations and requirements, indicates how each model element introduced in a specific diagram is tied to other elements defined in other diagrams, showing the existence of an underlying unique logical model for the whole system.

3.1 Requirements

Requirements diagrams are a new diagram type introduced by SysML. The language allows defining text based requirements and relations between requirements and the elements of the model. Requirements can specify capabilities related to structural elements, but also functionalities that must be provided or constraints that must be satisfied.

Requirements are described using a dedicated element "requirement" defined as a stereotype of a UML Class. Notice that the SysML language, being defined as an extension of a part of the UML meta-model, uses the same extension mechanisms provided by UML (e.g. tagged values, stereotypes and profiles). Each requirement element provides a textual description of the requirement and defines a unique identifier for the same.

Figure 3 shows the requirement diagram of the GRC problem. The requirement element labeled *RailroadCrossingSpecification* defines at a very high level of abstraction the goals of the system. Every other requirement in the diagram is related to this one. For instance the *RailroadCrossingSpecification* requirement is directly related to the *SafetyProperty* and *UtilityProperty* requirements, which describe properties that the whole model has to satisfy, while the *StructuralProperties* requirement provides an informal description of the system structure.

SysML provides different kinds of extensions of the UML association and dependency relationships for defining the relationships between couples of requirement elements, as. In particular SysML introduces a derivation relationship (as an extension of UML dependency) and a containment relationship (as an extension of UML association). The derive relationship *deriveReqt* is used to describe that a requirement may be derived from another one; however it has not a formal semantics and can be used to show extensions of properties, as between *OccupancyInterval* and *OccupancyDuration* elements. The containment relationship has been used among the *RailroadCrossingSpecification* and *StructuralProperties*, *SafetyProperty* and *UtilityProperty* requirements, but also among *StructuralProperties* and the other elements that describe the components of the system. This kind of relationship allows the decomposition of complex requirements into sets of simpler ones. Reading the text fields of requirements we find that the system is composed of a gate, a controller, *K* tracks and three sensors for each track. In this case requirements describe in an informal way the structure of the system, and suggest the organization of the architecture (which will be introduced later by means of other diagrams). We can also find a short description of the functionalities and features of each component. Each requirement can be related to either a specific model element or an aggregation of model elements. In fact, requirements can collect properties and constraints that will be referenced by different components. As an example, the *SafetyProperty* requirement defines a property related to the whole system and not to a single specific model element, while the *Gate* requirement specifies the existence of a unique physical gate component.

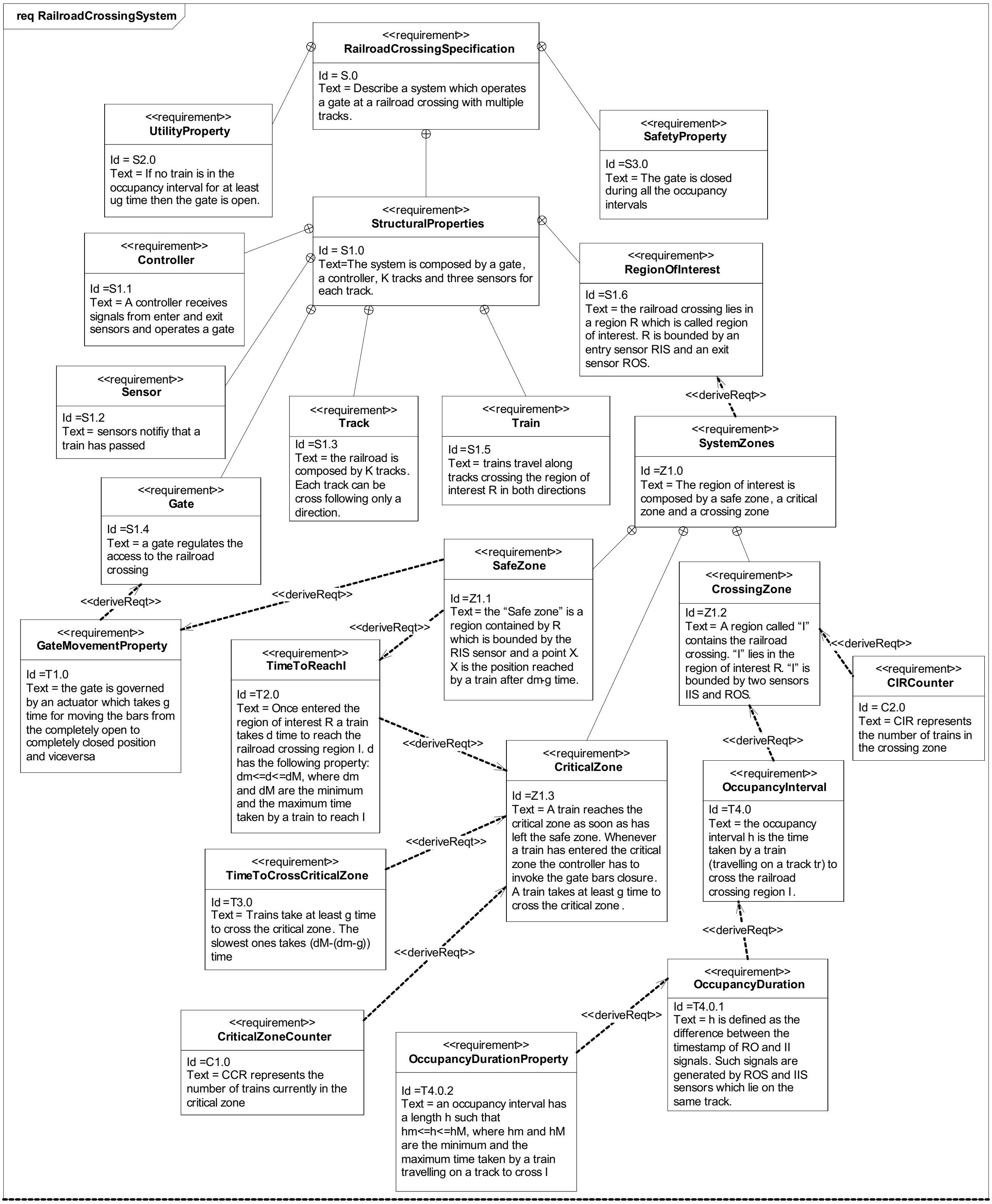

req RailroadCrossingSystem
<<requirement>> RailroadCrossingSpecification
Id = S.0
Text = Describe a system which operates a gate at a railroad crossing with multiple tracks.
<<requirement>> UtilityProperty
Id = S2.0
Text = If no train is in the occupancy interval for at least ug time then the gate is open.
<<requirement>> SafetyProperty
Id =S3.0
Text = The gate is closed during all the occupancy intervals
<<requirement>> StructuralProperties
Id = S1.0
Text=The system is composed by a gate, a controller, K tracks and three sensors for each track.
<<requirement>> Controller
Id =S1.1
Text = A controller receives signals from enter and exit sensors and operates a gate
<<requirement>> RegionOfInterest
Id =S1.6
Text = the railroad crossing lies in a region R which is called region of interest. R is bounded by an entry sensor RIS and an exit sensor ROS.
<<requirement>> Sensor
Id =S1.2
Text = sensors notifiy that a train has passed
<<requirement>> Track
Id =S1.3
Text = the railroad is composed by K tracks. Each track can be cross following only a direction.
<<requirement>> Train
Id =S1.5
Text = trains travel along tracks crossing the region of interest R in both directions
<<deriveReqt>>
<<requirement>> SystemZones
Id =Z1.0
Text = The region of interest is composed by a safe zone, a critical zone and a crossing zone
<<requirement>> Gate
Id =S1.4
Text = a gate regulates the access to the railroad crossing
<<requirement>> SafeZone
Id =Z1.1
Text = the "Safe zone" is a region contained by R which is bounded by the RIS sensor and a point X. X is the position reached by a train after dm-g time.
<<requirement>> CrossingZone
Id =Z1.2
Text = A region called "I" contains the railroad crossing. "I" lies in the region of interest R. "I" is bounded by two sensors IIS and ROS.
<<deriveReqt>>
<<deriveReqt>>
<<deriveReqt>>
<<deriveReqt>>
<<requirement>> GateMovementProperty
Id =T1.0
Text = the gate is governed by an actuator which takes g time for moving the bars from the completely open to completely closed position and viceversa
<<requirement>> TimeToReachI
Id =T2.0
Text = Once entered the region of interest R a train takes d time to reach the railroad crossing region I. d has the following property: dm<=d<=dM, where dm and dM are the minimum and the maximum time taken by a train to reach I
<<requirement>> CIRCounter
Id = C2.0
Text = CIR represents the number of trains in the crossing zone
<<deriveReqt>>
<<deriveReqt>>
<<requirement>> CriticalZone
Id =Z1.3
Text = A train reaches the critical zone as soon as has left the safe zone. Whenever a train has entered the critical zone the controller has to invoke the gate bars closure. A train takes at least g time to cross the critical zone.
<<requirement>> OccupancyInterval
Id =T4.0
Text = the occupancy interval h is the time taken by a train (travelling on a track tr) to cross the railroad crossing region I.
<<deriveReqt>>
<<requirement>> TimeToCrossCriticalZone
Id =T3.0
Text = Trains take at least g time to cross the critical zone. The slowest ones takes (dM-(dm-g)) time
<<deriveReqt>>
<<deriveReqt>>
<<requirement>> OccupancyDuration
Id =T4.0.1
Text = h is defined as the difference between the timestamp of RO and II signals. Such signals are generated by ROS and IIS sensors which lie on the same track.
<<deriveReqt>>
<<requirement>> CriticalZoneCounter
Id =C1.0
Text = CCR represents the number of trains currently in the critical zone
<<requirement>> OccupancyDurationProperty
Id =T4.0.2
Text = an occupancy interval has a length h such that hm<=h<=hM, where hm and hM are the minimum and the maximum time taken by a train travelling on a track to cross I

Figure 3. The requirements diagram for the railroad crossing problem

3.2 The structural aspects

The first step of the modeling process consists in defining the static aspects, i.e., the structure of our heterogeneous system. This activity is quite similar to the definition of an architecture for a software system [5], although here the subject is the problem domain.

SysML defines the Block element type as the elementary building block to describe a system structure. Each block defines a collection of features, properties, and operations that represent an element of the system. The system is modeled as a block that can be recursively decomposed into other blocks down to the basic elements of the structure. The resulting system structure is thus a hierarchy of blocks.

SysML provides Block Definition diagrams (BDD) and Internal Block diagrams (IBD) to define blocks, their internal structure and the relationships among them. Such diagrams are extensions of the UML Class diagram and Composite Structure diagram. Among the most innovative aspects introduced by these extensions there is a new kind of connection, based on flow ports through which material, energy, data or signals can flow (enter or leave the block); usually a flow port is intended to be used for asynchronous broadcast, or "send and forget" interactions. For instance in Figure 4 the Block *Sensor* exhibits a flow port whose type is *SensorData* and whose name is *sensP*.

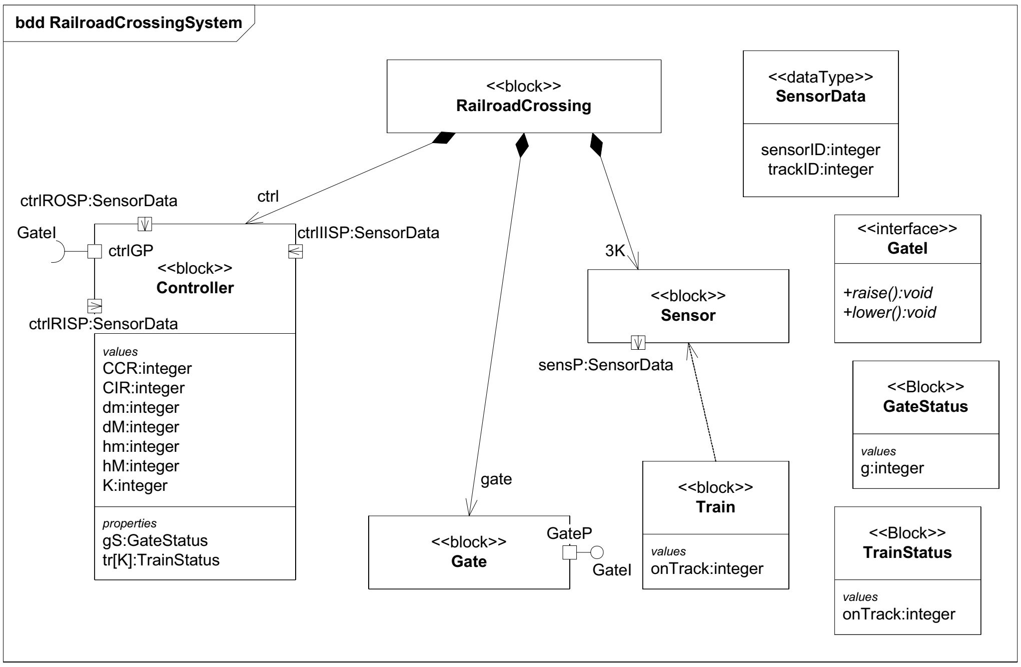


Figure 4. The Block Definition diagram for the railroad crossing system

We start the modeling activity defining the Block Definition diagram for the system (Figure 4). The system is composed of a block *RailroadCrossing*, which is decomposed into block *Gate*, *Controller* and *Sensor*.

The *Controller* block defines three input flow ports of type *SensorData* and a standard port labeled *ctrlGP*. A controller has three different input sources, one for each type of sensor. The port *ctrlGP* requires an interface *GateI*, which represents the interaction point with a gate component. Such interface defines two methods raise and lower that can be used to operate the gate. The *Controller* defines a set of attributes that keep track of the status and properties of regions. The *CCR* attribute represents the number of the trains currently in the critical region, while the *CIR* attribute represents the number of trains in the region of interest. The other attributes define constant values as the minimum or the maximum time to cross a specific region (as defined in the requirements diagram, see Figure 3) and the maximum number of tracks that compose the system. Therefore, the controller owns attributes of type *TrainStatus* and *GateStatus* that represent local models of the trains and of the gate, respectively. Such components are used in order to keep track of the state of the trains and of the gate. More in detail, block *TrainStatus* represents the position of trains in each track of the region of interest *R*. We are not interested in the exact position of trains; therefore the position is defined in an abstract way: out of region *R*, in *RI-X*, in *X-II*, in *II-RO*. The position of the train is determined according to the information provided by the sensors, and taking into account the flowing of time. According to the specification, there will be exactly *K* instances of this class, representing the trains traveling on the *K* tracks of *R*. Such a constraint is represented by specifying the number of instances of the attribute of type *TrainStatus* in the *Controller* block.

The *Gate* block implements the *GateI* interface and therefore it defines how the raise and lower operations must be realized.

The *Sensor* block is characterized by an output flow port of type *SensorData*. A sensor will generate *SensorData* signals, which will be sent through this port. A *SensorData* signal contains the necessary attributes for storing information on the sensor that has generated signal.

It can be noticed that the proposed model adopts some simplifications: for instance, *dm*, *dM*, *hm*, *hM* should be functions of both train speed and region lengths. Moreover blocks representing tracks and regions are missing. We decided not to represent these items in order to keep the model as simple as possible, while focusing on the specification of the real-time behavior of the system. It is easy to enhance the model in order to include all the omitted details, and in order to reflect more faithfully the structure of the real system. Despite this simplification, the structure directly satisfies what was defined in the requirements diagram. For instance, tracks do not play a fundamental role in the system, since they represent a collection of sensors and trains. Trains traveling along a track interact with sensors that generate signals that will be captured by the controller. Block *Train* owns an attribute that identifies the track on which the train is traveling. The same applies to the *Sensor* block.

An Internal Block diagram is used to show more in detail how blocks are interconnected.

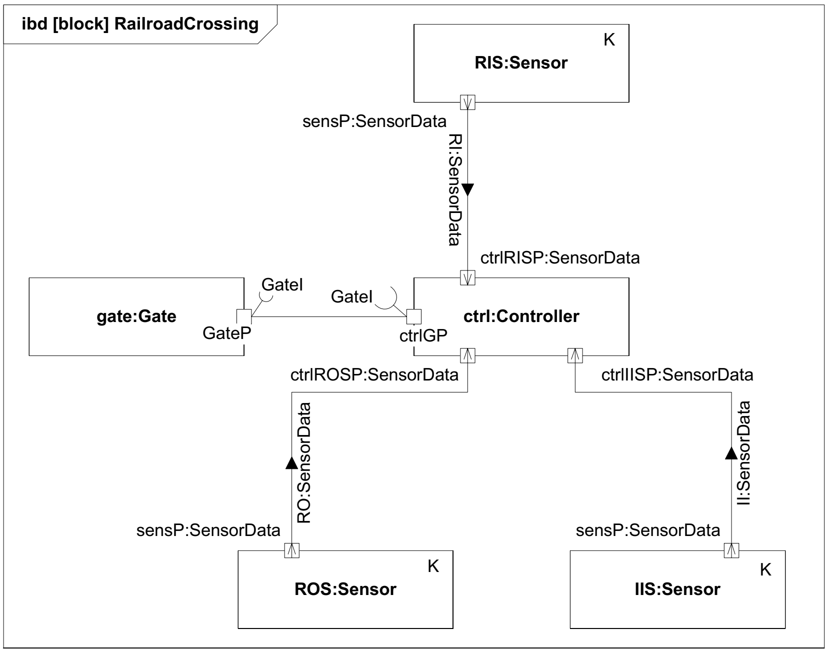


Figure 5. The Internal Block diagram for the block RailroadCrossing

In Figure 5 we can see a sketch of the structure of the system. This diagram defines the instances of the blocks defined in the Block Definition diagram. Notice that the same names of instances appear in different diagrams. Here we find a first usage of the model centric vision introduced above and a first application of separation of concerns principle. We defined a model element in a dedicated diagram and then used the same element in a different diagrams.

The *RailroadCrossing* block is composed of different parts. A *Gate* instance labeled *gate* is directly connected to *ctrl*, an instance of the *Controller* block. The connection is specified by means of a usage relationships between elements that require and provide the interface *GateI* (see the Block Definition diagram in Figure 4). Three instances of the *Sensor* block are connected to *ctrl* through flow ports of type *SensorData*. More in detail, for each track i (with 1<=*i*<=*K*) we have three sensors labelled *RIS*[*i*], *ROS*[*i*] and *IIS*[*i*]. *RIS*[*i*] is the sensor at the beginning of the region of interest *R*, *IIS*[*i*] is the sensor at the beginning of the crossing zone *I* and *ROS*[*i*] is the sensor at the end of the crossing zone (and of the region of interest too) of the *i*-th track. Each sensor will produce *SensorData* signals to indicate on which track the sensor operates and the position of the sensor on the track. For example the *RIS*[*i*] sensor will generate a *SensorData* signal characterized by a attribute *trackID=i* value and a attribute *sensorID=RIS* (indicating that the sensor is placed at the entrance of the safe region).

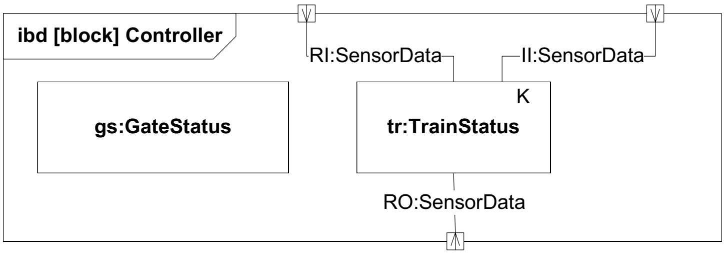


Figure 6. The Internal block diagram for the *Controller* block

Figure 6 describes the architecture of the *Controller* block. The *Controller* is composed of an instance of *GateStatus* that stores the current position of the bar of the gate, and by *K* instances of *TrainStatus* that keep track of the position of the trains. Each *TrainStatus* object receives signals from the sensors and changes its state if it is on the same track as the sensor.

3.3 Abstract dynamics: Activity diagrams

The modeling of the system goes on with the definition of the abstract dynamics of the system. For this purpose SysML extends UML activity diagrams with constructs aimed at handling continuous flows of data (streaming activities). For more detailed information see [6].

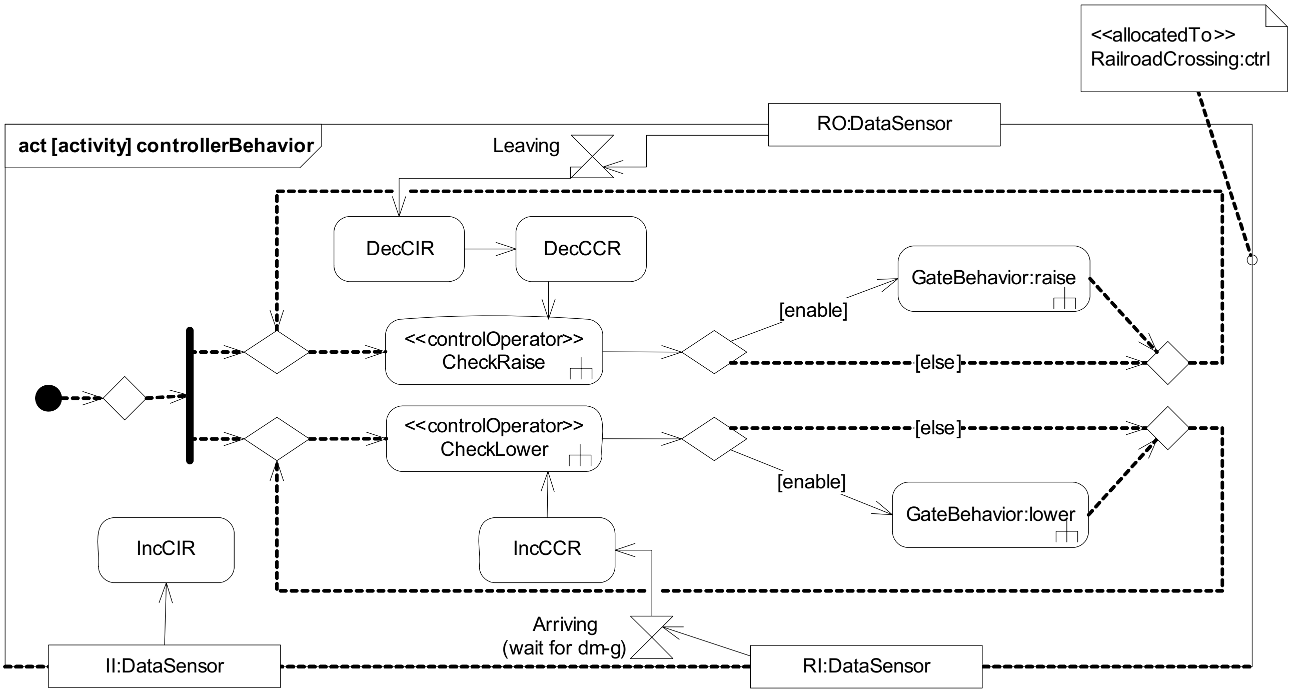


Figure 7. *Controller* behavior

The main activity of the system is shown in Figure 7. The squared boxes on the border of the diagram represent incoming data from the 3 sensors flow ports ctrlIISP, ctrlRISP and ctrlROSP (see the Internal Block diagram in Figure 5); in fact each of these ports is connected to K sensors (one sensor for each track).

The effect of the signals from sensors on the status of the controller is specified by the activities DecCIR, DecCCR, which decrement counters CIR and CRR respectively, and by activities IncCIR and IncCCR, which increment them. Counters CIR and CCR (that are part of the state of the Controller, see Figure 4) represents the number of trains in the critical region and in the crossing region, respectively.

The values of these attributes have to change only according to trains entering or exiting the znes of the interest region. As an example, consider the entrance in the critical region. There are not sensors that notify to the controller that a train has entered such zone, but the controller assumes that a train is entering *(dm-g)* seconds after the entrance in the interest region. Therefore the diagram shows that the execution of IncCCR is strictly triggered by the receipt of data from the RI sensor, with a (dm-g) delay. The activation is labeled “Arriving” in order to synchronize the dynamic behaviour specified by the activity diagram with the behaviour specified by state diagrams (Leaving is used in the state machine of the Controller, Figure 12).

The dynamics of *TrainStatus*, *GateStatus* and *Controller* will be further refined in the State diagrams of Figure 10, Figure 11 and Figure 12.

The data coming from sensors RO and RI enable the activities *CheckRaise* and *CheckLower* respectively. These activities check the values of the counters to decide whether the gate has to be lowered or raised. Then the control returns to the initial point for a new iteration of the control loop. We can see clearly in this diagram that the flow of actions is not only specified by the control flow (marked as a dotted line), but is also influenced by incoming data (solid line).

An activity can be further described by an Activity diagram showing inner activities and flows. An example of such descriptions is shown in Figure 8, where activities *CheckRaise* and *CheckLower* are described. In both the diagrams the Control Value data explicitly model the outputs of the activities used to decide whether to enable *raise* and/or *lower*.

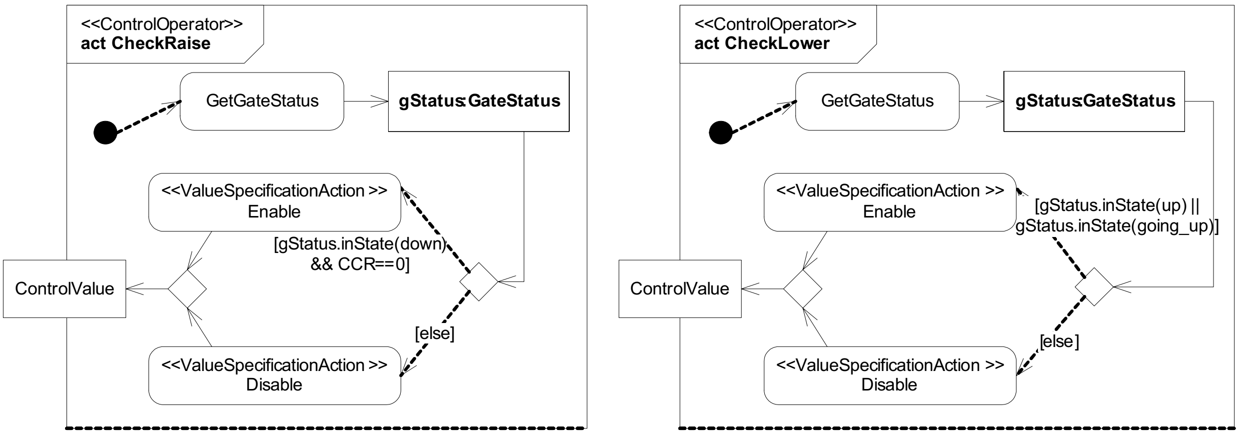


Figure 8. *CheckRaise* and *CheckLower* activities behaviors

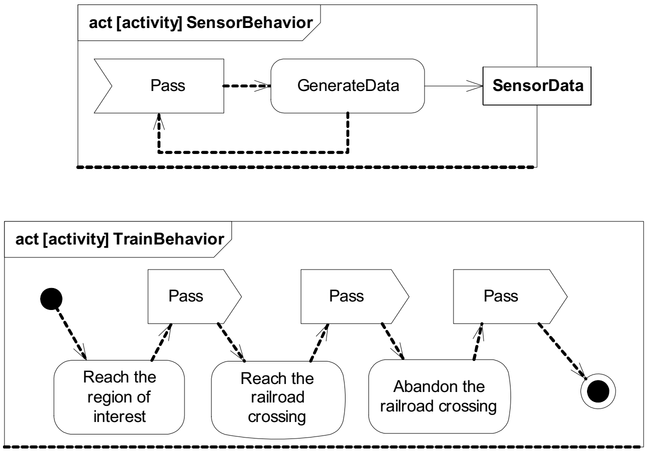


Figure 9. Train and Sensor Behavior

The remaining Activity diagrams of our model are simpler and they are shown in Figure 9. The train activities simply consist in moving through the region of interest and the crossing zone; each time the train goes through a sensor an event is fired to let the proper sensor generate data and make it available to the rest of the system.

3.4 State diagrams

SysML provides state machine diagrams that inherit all the features of the corresponding UML diagrams. State machines can be used to describe the behavior of an element of a system in terms of its states and transitions.

In Figure 10 we can see the state machine diagram of the block *TrainStatus*. The block definition diagram in Figure 4 shows that a train is characterized by a property *onTrack* that specifies the name of the track on which the train is traveling. Let us consider a train traveling on a track *Tr*. Initially the train is out of the region of interest, then it reaches such region (coherently with the activity diagram of Figure 9) and passes the sensor *RIS*. Such sensor sends the signal *RI*, which is an instance of the block *SensorData* with *sensorID* set to *RIS* and *trackID* equal to *Tr*. When the sensor sends its signal, the train is entering the safe zone and after *dm-g* seconds the fastest trains are entering the critical zone (at point *X*). This transition is governed by a time-out, modeled by means of the *after* statement provided by UML [10] and inherited by SysML.

Now we should specify that the train enters region *I* not earlier than *g* seconds nor later than *dM*-(*dm-g*) seconds after entering the critical zone. SysML does not provide any means to specify that a transition cannot happen before a given time or must occur before a given time. In order to express such constraints we had to introduce the state *Close_to_Crossing* and the state *Error*. Notice that these states were introduced only because of the limits of the language. Notice also that the state *Error* indicates that the behavior of the system deviated from the specification. In designing a real system, we would wish to substitute the state *Error* with some exception handling state, for the sake of robustness.

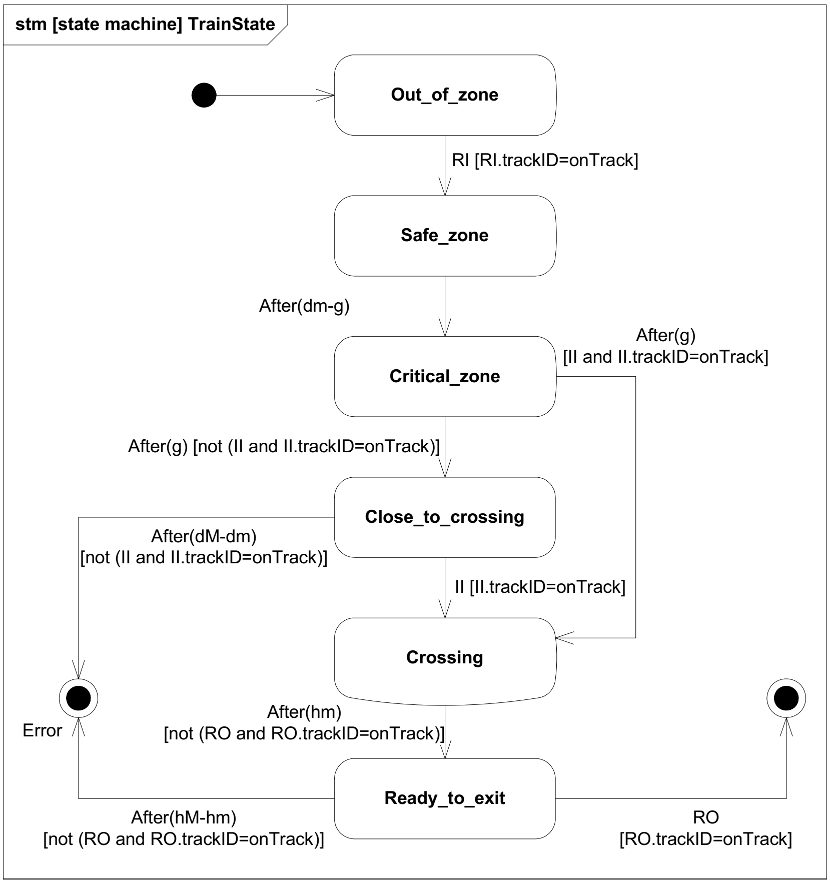

stm [state machine] TrainState
Out_of_zone
RI [RI.trackID=onTrack]
Safe_zone
After(dm-g)
Critical_zone
After(g)
[II and II.trackID=onTrack]
After(g) [not (II and II.trackID=onTrack)]
Close_to_crossing
After(dM-dm)
[not (II and II.trackID=onTrack)]
II [II.trackID=onTrack]
Crossing
Error
After(hm)
[not (RO and RO.trackID=onTrack)]
Ready_to_exit
After(hM-hm)
[not (RO and RO.trackID=onTrack)]
RO
[RO.trackID=onTrack]

Figure 10. State machine diagram for the *TrainStatus* block

Consider now how to specify what happens if a train reaches point *II* exactly after it has been in the Critical zone for *g* seconds (that can actually happen, for the fastest trains). In such a case the train goes directly into the Crossing state, without passing into the state *Close_to_Crossing*. In order to express this constraint we borrowed from UML the syntax of conditions, extending their semantics to express the occurrence of an event at a given time (in plain UML conditions cannot contain references to events). The meaning of the remaining transitions should be clear, thus they are not commented here.

The State diagram of the block *Gate* is shown in Figure 11.

Most transitions are easy to understand, and modeling them presented no difficulty. When the gate is closed (i.e., the bar is down), and a raise command is received, the bars start to move upwards. If a lower signal occurs when the gate is still opening, the bars must start to move down immediately: this is modeled by a transition to state *InvertedDown*. According to the problem definition the bar will reach the closed position after a time equal to the time it has been opening. In order to represent this behavior, we have to explicitly refer to the time when such event occurred: transition to state *Down* will occur after a time equal to the interval between the last open and close commands. When the gate is in the state *Up*, and a lower command is received, the bars start to move downwards. If a raise signal occurs when the gate is still closing, the bars must start to move up. This is described by a transition to state *InvertedUp*.

The states *InvertedUp* and *InvertedDown* require some comments. The state *InvertedDown* can be reached as a consequence of a lower command sent by the controller whenever a train reaches the Critical region while the gate is opening. The state *InvertedUp* should never be reached as a consequence of a command sent by the controller (see the state machine diagram for the *Controller* block). Consider such diagram as the description of the generic behavior of a gate.

By allocating the states and the transitions of this machine to the instances of the blocks, signals and activities of our system, we define which transitions can fire and therefore which states can be reached.

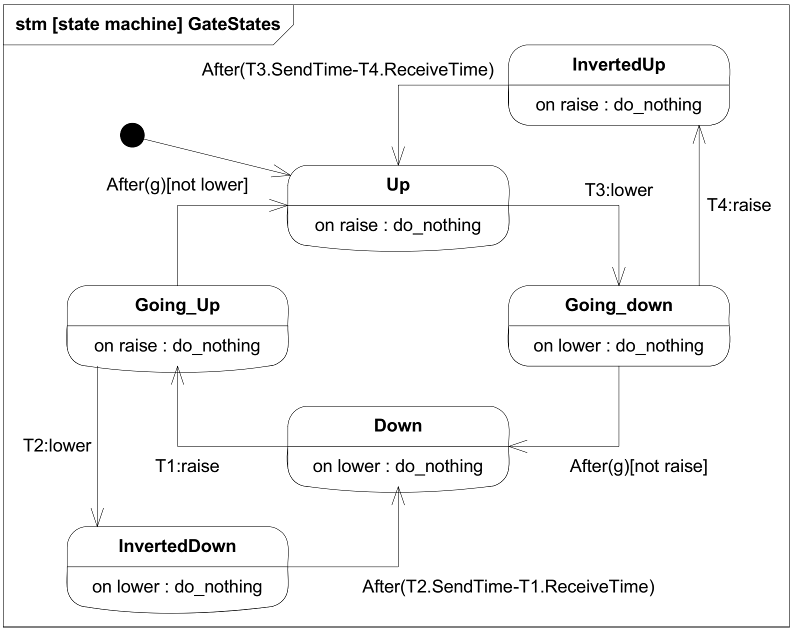


Figure 11. State machine for the block *Gate*

SysML provides the keywords *ReceiveTime* and *SendTime* to indicate the time when an event was issued and received, respectively. However, we also need to indicate to which of the raise or lower signal we refer (e.g., we are interested in the lower event occurred while the gate was in state *Going_Up*, not to the one occurred while in Up state). The language provides a labeling mechanism that can be used to identify individual transition *instances*. State diagrams represent transition *types*, i.e., every transition may occur several times, that is, it may have several instances. We thus extend SysML syntax and semantics by applying labels to transitions in the state diagrams, so that we always refer to the last occurrence. Thus, the transition from *InvertedDown* to *Down* occurs after a period equal to the interval between the last raise command and the last lower command.

The state diagram of the *Controller* block is reported in Figure 12.

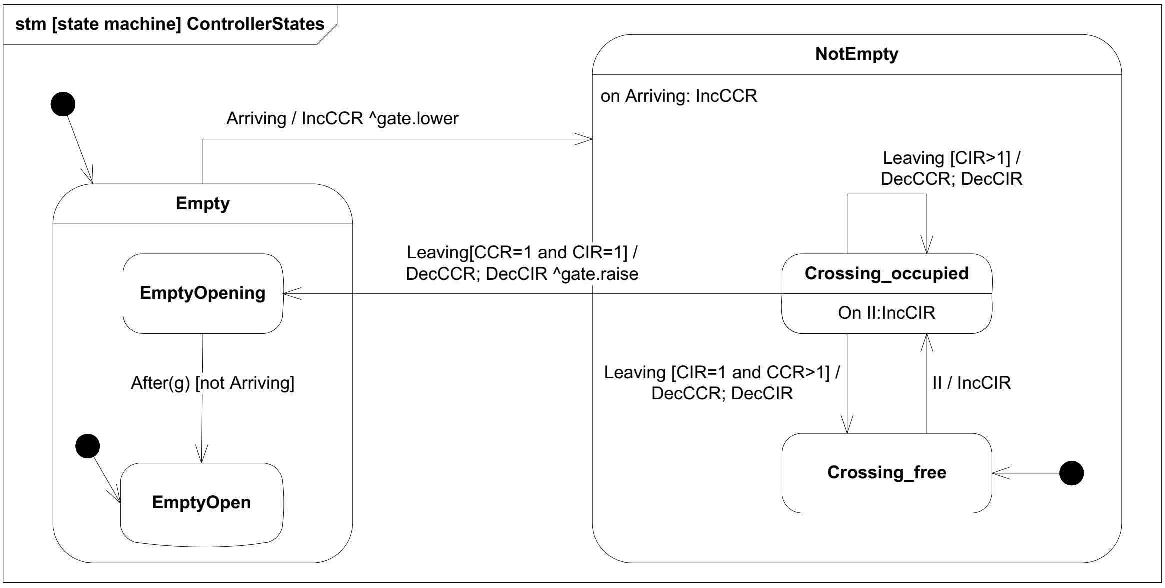

stm [state machine] ControllerStates
NotEmpty
on Arriving: IncCCR
Arriving / IncCCR ^gate.lower
Leaving [CIR>1] / DecCCR; DecCIR
Empty
Leaving[CCR=1 and CIR=1] / DecCCR; DecCIR ^gate.raise
EmptyOpening
Crossing_occupied
On II:IncCIR
After(g) [not Arriving]
Leaving [CIR=1 and CCR>1] / DecCCR; DecCIR
II / IncCIR
Crossing_free
EmptyOpen

Figure 12. The state machine diagram for the Controller block

Initially the *Crossing* is empty and the gate is open (this means that there are no trains in the critical zone). Whenever a train enters the critical zone the counter *CCR* is incremented invoking the *IncCCR* activity. Whenever a train exits the critical zone the counter *CCR* is decremented. If a train arrives while the critical zone is empty the state is changed to non-empty, and a close command is sent to the gate. When the last train leaves the critical zone, the state is changed to empty, and an open command is sent to the gate.

3.5 Expressing model properties

SysML provides constraint blocks to define properties and constraints on elements of the model. Constraint blocks enable the application of engineering analysis mechanisms to check the consistency of the system. A Constraint block is an extension of a standard block; it is characterized by a property and a set of parameters that represent the basic elements involved by the property. SysML provides a mechanism to separate the definition of a constraint element from its usage. A constraint block specifies a context-independent property. Defined constraints can be reused involving different elements of the same model. Parameters are the binding points among the elements of the model and the property expressed by the constraint. Block constraints are specified using block definition diagram. SysML also provides Parametric diagrams, an extension of the Internal block diagram that can be used to define the usage of constraints and thus to define instances of constraints elements. Such diagrams provide constructs to apply the binding mechanism.

Different languages can be adopted to define constraints using different levels of rigor, e.g., according to the criticality of the system. In particular, formal languages can be applied whenever it is advisable to define crucial properties that have to be assured. As a consequence we could say that SysML provides an easy way to apply “lightweight” formal methods, supporting rigor and formality when needed.

As we can see in the Internal Block diagram in Figure 5, the *Controller* block has only one instance, which represents the current situation and the criteria to be followed in sending commands to the gate. The controller must always know how many trains are in the critical and in the crossing zones. Thus, in the block definition diagram defined in Figure 4 we have introduced the attributes *CCR* and *CIR*, which count how many trains are in these regions. Both counters are initially set to zero, and are modified by the increment and decrement methods.

We use the Object Constraint Language (OCL) [12] for defining constraints on SysML blocks. For this purpose SysML blocks are treated as UML Classes and activities allocated to a block are treated as operations provided by a class. Here follows a selection of the properties related to the *Controller* block:

```
context Controller inv :
    self.CCR>=0
```

```
context Controller::IncCCR inv :
    CCR@pre<K and CCR=CCR@pre+1
context Controller::DecCCR inv :
    CCR@pre>0 and CCR = CCR@pre-1
context Controller inv :
    self.CIR>=0 and self.CIR<=self.CCR
context Controller::IncCIR inv :
    CIR@pre<K and CIR=CIR@pre+1
context Controller::DecCIR inv :
    CIR@pre>0 and CIR = CIR@pre-1
```

Constraints concerning the minimum and maximum traversing times are expressed by means of the following OCR statement:

```
context Controller inv :
   Self.dM >= Self.dm
   Self.dm > Gate.g
   Self.hM >= Self.hm
   Self.hm > 0
   Self.dm > 0
```

The GRC can be classified as a real-time safety critical system. As a consequence, we have decided to use the TRIO [21] formal language for the definition of the constraints on our model. TRIO is a first order logic language augmented with a temporal domain, arithmetic operators and temporal operators. Such language has been applied as a specification language to different industrial case studies [20] and it is well suited to describe the most crucial properties related to our problem.

We consider the same invariant properties for the *Controller* block that we described above with OCL. Figure 13 reports the definition of such properties using constraint blocks in a block definition diagram and using TRIO as specification language.

As an example, let us consider the constraint *IncCounterInvariant*. Such block defines a property related to three different parameters. The property states that given a value *C* and a signal *IncC*, for each k such that up to now *C* was equal to *k*, when *IncC* occurs *C* will be equal to *k*+1. Then the property says that a time instant exists such that *C* is equal to 0 and before such instant *C* was always equal to 0. A detailed explanation of TRIO formulas and temporal operators does not fit in the objectives of this paper and can be found in [21]. A brief description of the most common TRIO operators is reported in the appendix.

Let us consider the Block definition diagram shown in Figure 13, in which all the defined constraint blocks are apparently independent from the other elements of the model. For instance the constraint *DecCounterInvariant* can be considered as a generic property that can be instantiated and bound to different couples of elements of the model matching with the *DecC* and *C* parameters of the constraint block.

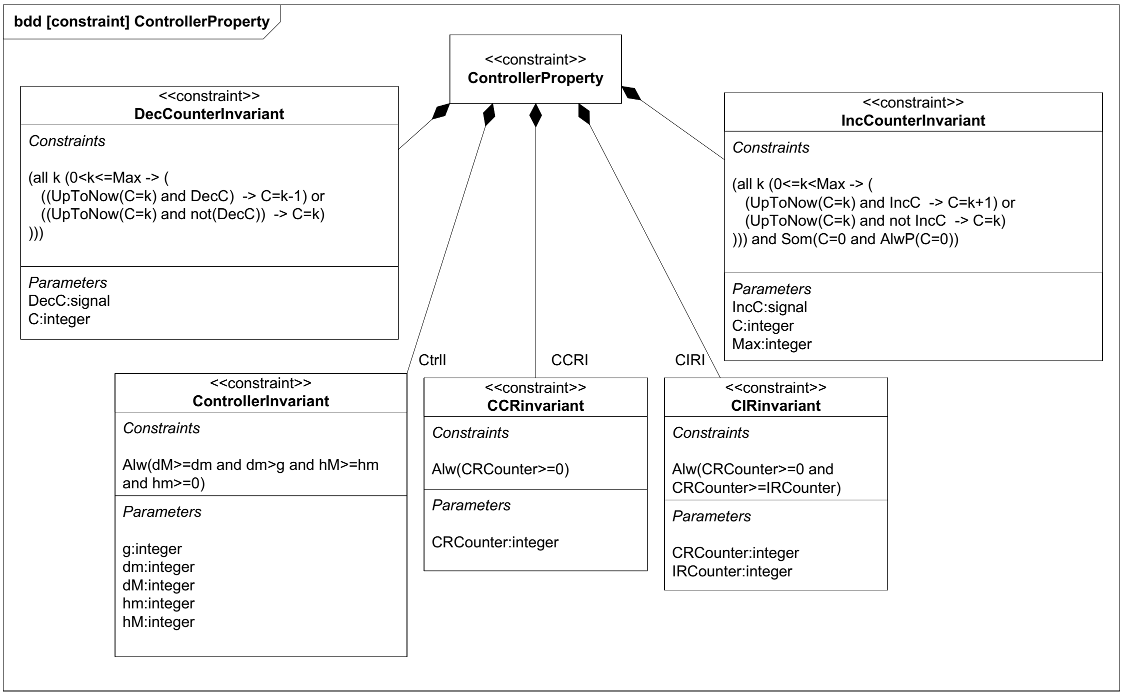


Figure 13. Definition of invariant properties for the Controller block

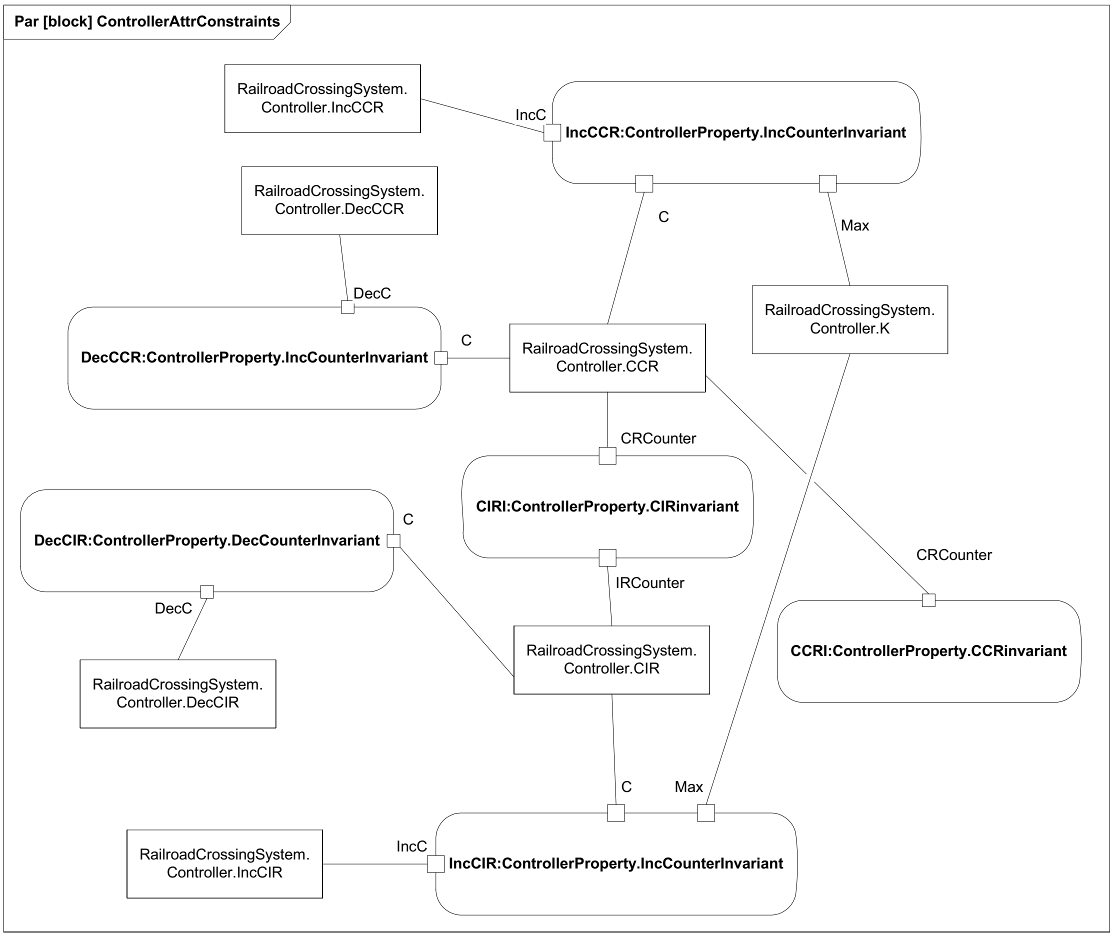


Figure 14. The parametric diagram which shows how the instances of the properties for the Controller block are bound to other element instances of the GRC system.

Figure 14 reports a Parametric diagram that specifies how the parameters of the constraints defined in Figure 13 are allocated to instances of elements. For instance, parameter *C* of the *DecCounterInvariant* constraint is allocated to the *CIR* attribute of the *Controller*.

Although the model discussed so far is sufficient to guarantee that the gate controller works properly, it is not detailed enough to express the safety property. In fact, we need to say that "when there is at least one train in the crossing zone *I*, the gate is closed". Thus we need to refine what we have defined in the requirements diagram using a formal notation. The safety condition can be written in OCL as follows:

```
Context Controller inv:
    Self.oclInstate(Crossing_occupied) implies Gate.oclInstate(Down)
```

Expressing the utility property involves some problems. Exploiting the state diagrams, it is only possible to state the following property:

```
Context Crossing inv:
    Self.oclInstate(Empty) implies
    Gate.oclInstate(Going_Up)or Gate.oclInstate(Up)
```

However, it is easy to see that the property above is too weak, as it does not require the gate to eventually reach the open position. The utility property, instead, requires that the gate is open whenever the crossing has been empty for at least *g* seconds (i.e., for the time required to open the gate):

```
Context Crossing inv:
    Self.oclInstate(Empty) for at least g implies Gate.oclInstate(Up)
```

Unfortunately, the sentence above is not a legal OCL statement, since OCL does not provide a way to deal with time. Instead, we can use a dedicated language as TRIO to express both properties.

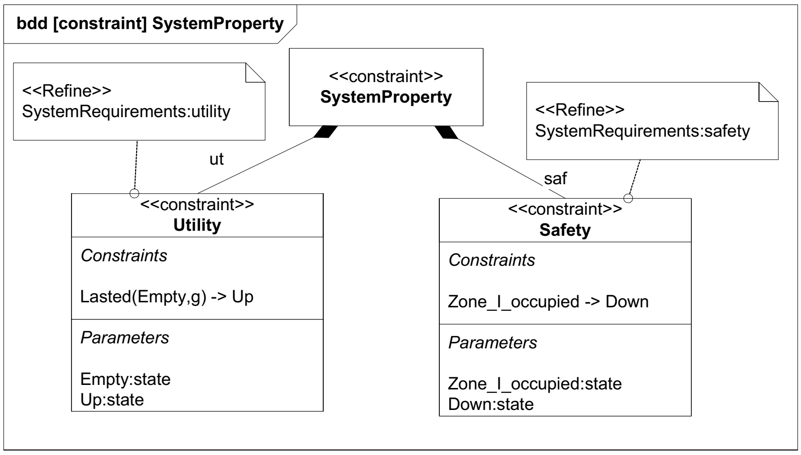


Figure 15. Expressing the Utility and Safety properties

As an example let us consider the constraint *Utility* defined in Figure 15. It uses a couple of parameters and a property expressed in TRIO. The property states that after the state *Empty* lasted for *g* time units the state is *Up*. This expression specifies the property that we could not express using OCL. It is now necessary to associate it with the elements of the model. When instantiating the property we state that the parameter *Empty* is associated with the state *Empty* of the state machine diagram shown in Figure 6, and, as a consequence, it represents that no train is in the crossing region. Similarly the parameter *Up* is associated with the state *Up* of the state machine that describes the behavior of the block *Gate*.

The constraint blocks *Utility* and *Safety* are introduced in Figure 15 as refinements of the informal properties described in the requirements diagram shown in Figure 3. In this way the user requirements –which are often defined in natural language– are complemented by constraint blocks that provide a formal specification of the critical system properties.

4 A FIRST ASSESSMENT

SysML represents a big step forward in modeling systems; it implements most of the requirements found in the original request of proposals submitted to OMG in 2003 (see [13]), often enhancing them. Flow ports and continuous activities have been introduced to adequately address the modeling of continuous systems [6], thus overcoming a limit of UML 2.0.

Considering embedded and real-time systems, it is quite clear that the capabilities of SysML to model such systems are far better than UML 2.0 ones. As an example think about the problem of modeling a control system, a type of system that usually has both real-time and embedded features. Notice that the communications among the components of such systems usually require continuous flows of information, and while SysML provides constructs to describe streams of data and allows one to deal with them (as described in [6]), UML does not allow modelling continuous data flows. Nevertheless, some needed features are poorly supported by SysML or are completely missing.

SysML provides the extension mechanisms that were inherited from UML (tagged values, stereotypes and profiles). Such extensions are useful to adapt the language to describe a specific problem domain. However, these extension mechanisms are not powerful enough to change the semantics of the basic constructs. As an example, consider the extensions defined for the state machine diagrams in [19] or for the sequence diagrams in [23]. The mechanisms provided by UML are not powerful enough to change the semantics of the transitions in the state machine diagrams and of the interactions in the sequence diagrams, so in both cases changes must be defined by directly manipulating the UML meta-model.

For the GRC problem we have preferred not to use a profile (e.g., a temporal profile) to model the system. Instead, we have based our modeling activity on the usage of the basic profile of SysML. There are two main reasons for this choice: one is that at the moment no timing profiles are available for SysML, the second is that we were interested in studying the basic capabilities of SysML and how the usage of a formal notation could be merged in the modeling process.

Focusing the attention on the usage of the standard constructs and diagrams we have found the inadequacy of the language to address the problem of modeling strict timing information. In fact, the state diagram behavior of SysML is the same of UML 2.0: it follows a run-to-completion semantics. The run-to-completion semantics provides each state machine with a buffer to handle the incoming events; every event is then consumed when the machine has finished consuming the previous one. The consumption of an event can trigger a transition that executes in a non null time. While this behavior is acceptable in modeling the implementation of a system, it is unsuited to model the time requirements and to carry out the analysis of a system. For this purpose instantaneous and possibly simultaneous transitions are needed, as well as the ability to deal with time intervals. Timed Statecharts [22] are suitable to express precise time constraints (see [15], [16] and [19] where the problem is specifically addressed). Besides, syntax and graphical constructs of Timed Statecharts are very similar to analogous constructs found in SysML (UML 2.0). Therefore, the state diagrams presented above model well the time behavior of the system if we assume that they follow the Timed Statecharts semantics.

SysML does not feature a proper temporal logic (or analogous formalism) to express time constraints. Although SysML does not prevent the use of any formal language to express constraints, the availability of a standard formal language would be useful, especially considering that UML constraint language OCL does not support the specification of timing issues, and it lacks many of the needed features (see [18] for an OCL proposal of extension with time related features). In practice, modelers have to employ a language like TRIO [21] to represent time constraints.

A final remark is that resource management and consumption –which are very important features when dealing with embedded systems– are not directly addressed. On the positive side, the buffered behavior of each activity in Activity diagrams can be disabled in SysML, in order to model activities that do not buffer incoming data. In SysML it is even possible to constrain the rate of incoming or outgoing data that activities can accept (thus preventing buffering). These features are relevant for embedded systems since buffers can consume large amount of resources.

5 CONCLUSIONS

Our goal was to apply and assess a full model-centric approach to the definition of a complex heterogeneous system using SysML, in order to evaluate the capabilities of ysML by applying it to the analysis and design of a system characterized by real-time safety critical requirements. We chose the GRC problem as a reliable and well-known benchmark for this kind of systems.

We found SysML well suited to support a model-centric development, since it provides dedicated constructs to describe all the aspects of a system: it supports the definition of both structural and behavioral features, and provides constructs to describe and organize the requirements. The language also provides cross cutting constructs that allow the modeler to allocate behavioral elements on structural ones and vice-versa, and to relate the elements of the model to the requirements. Despite these remarkable features, the language is not fully satisfactory for modeling precise time requirements because the language inherits some of the weaknesses found in UML 2.0. SysML provides mechanisms to define extensions to its meta-model; constructs like tagged values, constraints, stereotypes and profiles can be used to extend the expressiveness of the language and to adapt the language to describe systems of a specific domain. Such mechanisms help to extend the capabilities of SysML, but do not allow redefining the semantics of its basic constructs. As a consequence, we were obliged to use external formal languages to address such issues. The usage of formal language like TRIO allows to express time properties and behaviors in a rigorous way and to possibly integrate the automated or semi-automated engineering analysis tools.

In the future we plan to apply SysML to a complex industrial case study in order to evaluate the scalability of this approach. Moreover we are interested in evaluating the consequences on the following development phases. As a second line of interest, we are working on coupling SysML with formal languages such as TRIO in order to come up with an effective development method for real-time embedded systems.

6 REFERENCES

[1] Douglass B. P. Real-Time UML, Addison Wesley, 1998.

[2] Heitmeyer C.L., Jeffords R.D., Labaw B.G., “Comparing different approaches for Specifying and verifying Real-Time Systems”, in Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software (New York May 1993), 122-129.

[3] C.L. Heitmeyer and N. Lynch. “The generalized railroad crossing: A case study in formal verification of real-time systems”, in Proc. of the IEEE Real-Time Systems Symposium, San Juan, Puerto Rico, December 7-9, 1994

[4] Selic B., Gullekson G., Ward P.T., “Real-Time Object-Oriented Modeling”, Wiley, 1999.

[5] Garlan D., Shaw M., “Software Architecture”, Prentice-Hall, 1996.

[6] Bock C., SysML and UML2 Support for Activity Modeling, Systems Engineering, Vol. 9, No. 2, Wiley, 2006.

[7] OMG, “Meta Object Facility (MOF) 2.0 XMI Mapping Specification”, v2.1 formal/05-09-01, September 2005

[8] OMG, “OMG Systems Modeling Language (OMG SysML) Specification”, Final Adopted Specification ptc/06-05-04, May 2006.

[9] OMG, “OMG Systems Modeling Language (OMG SysML) Tutorial”, July 2006.

[10] OMG, “Unified Modeling Language: Superstructure”, version 2.0, formal/05-07-04, August 2005.

[11] OMG, “Unified Modeling Language: Infrastructure”, version 2.0, formal/05-07-05, March 2006.

[12] OMG, “OCL 2.0 Specification”, version 2.0, ptc/2005-06-06, June 2005.

[13] SE-DSIG (OMG Systems Engineering Domain Special Interest Group), “UML for systems engineering RFP”, http://www.omg.org/cgi-bin/doc?ad/03-03-41, March 2003.

[14] L. Lavazza, G. Quaroni, M. Venturelli, “Combining UML and formal notations for modelling real-time systems”, Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), Vienna, 10-14 September 2001.

[15] V. del Bianco, L. Lavazza, M. Mauri, “A Formalization of UML Statecharts for Real-Time Software Modeling”, The Sixth Biennial World Conference on Integrated Design Process Technology (IDPT 2002), Pasadena, California, 23-28 June 2002.

[16] V. del Bianco, L. Lavazza, M. Mauri, “Model Checking UML Specifications of Real-Time Software”, The Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland, 2–4 December, 2002.

[17] L. Lavazza e G. Occorso "Simulation-based Verification of UML models", 15th International Conference on Software Engineering and Knowledge Engineering (SEKE2003), San Francisco, July 2003.

[18] L. Lavazza, S. Morasca, A. Morzenti, "A Dual Language Approach to the Development of Time-Critical Systems with UML" TACoS (International Workshop on Test and Analysis of Component Based Systems) in conjunction with ETAPS 2004, Barcelona, march 27 - 28, 2004. Electronic Notes in Theoretical Computer Science 116 (2005), 19 January, p. 227–239.

[19] V. Del Bianco, L. Lavazza, M. Mauri, G. Occorso, "Towards UML-based formal specifications of component-based real-time software", International Journal on Software Tools for Technology Transfer (STTT) - Springer-Verlag GmbH, online at: http://dx.doi.org/10.1007/s10009-006-0024-8.

[20] Ciapessoni E., Coen-Porisini A., Crivelli E., Mandrioli D., Mirandola P., Morzenti A., "From formal models to formally-based methods: an industrial experience", ACM Transactions on Software Engineering and Methodology, vol. 8. no 1, January 1999, pp.79-113.

[21] Ghezzi C., Mandrioli D., Morzenti A.. "TRIO a Logic Language for Executable Specifications of Real-time Systems", Journal of Systems and Software, vol. 12, n. 2, May 1990.

[22] Y. Kesten, A. Pnueli, "Timed and Hybrid Statecharts and their Textual Representation", Weizmann Institute of Science, In Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium, 1992.

[23] M. Elkoutbi, M. Bennani, R. K. Keller, M. Boulmalef, "Real-time System Specifications based on UML Scenarios and Timed Petri Nets", In International Workshop on Communication Software Engineering (IWCSE'2002), IEEE 2nd International Symposium on Signal Processing and Information Technology, Marrakech, Morocco, December 2002, pp 362-366

7 APPENDICES

7.1 Introduction to TRIO

TRIO is a first order temporal logic language that supports a linear notion of time [21]. Besides the usual propositional operators and quantifiers, one may compose formulas by using a single basic modal operator, called *Dist*, that relates the current time, which is left implicit in the formula, to another time instant: the formula *Dist(F, t)*, where *F* is a formula and *t* a term indicating a time distance, specifies that F holds at a time instant at *t* time units from the current instant.

A number of derived temporal operators can be defined from the basic *Dist* operator through propositional composition and first order quantification on variables representing a time distance.

Table 1. Some TRIO Temporal Operators definitions

Operator	**TRIO Definition**
Som(F)	(∃d)(Dist(F,d))
Alw(F)	¬Som(¬F)
Lasts(F,t)	(∀d)(0<d<t → Dist(F,d))
Lasted(F,t)	(∀d)(0<d<t → Dist(F,-d))
Until(F,G)	(∃d)(d>0 ∧ Lasts(F,d) ∧ Dist(G,d))
Since(F,G)	(∃d)(d>0 ∧ Lasted(F,d) ∧ Dist(G,-d))
UpToNow(F)	(∃d)(d>0 ∧ Lasted(F,d))
NowOn(F)	(∃d)(d>0 ∧ Lasts(F,d))

Table 1 reports the formal definition of some TRIO derived operators. Most of the operators are symmetrically defined with reference to the past and the future of the current instant. TRIO is well suited to deal with both continuous and discrete time.