5th International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES^MB 2012)

Preliminary proceedings

September 30th, 2012, Innsbruck, Austria

Co-organized by :

Iulian Ober, University of Toulouse-IRIT, France (*chair*)
Stefan Van Baelen, K.U.Leuven-DistriNet, Belgium
Thomas Weigert, Missouri S&T, USA
Sébastien Gérard, CEA-LIST/LISE, France
Huascar Espinoza, Tecnalia-ESI, Spain

Organizing Committee

- Iulian Ober, University of Toulouse-IRIT, France (*chair*)
- Stefan Van Baelen, K.U.Leuven-DistriNet, Belgium
- Thomas Weigert, Missouri S&T, USA
- Sébastien Gérard, CEA-LIST/LISE, France
- Huascar Espinoza, Tecnalia-ESI, Spain

Program Committee

- Nicolas Belloir, LIUPPA, France
- Jean-Michel Bruel, University of Toulouse-IRIT, France
- Agusti Canals, CS-SI, France
- Arnaud Cuccuru, CEA-LIST/LISE, France
- Jean-Marie Farines, UFSC, Brasil
- Peter Feiler, CMU-SEI, USA
- Mamoun Filali, University of Toulouse-CNRS-IRIT, France
- Sébastien Gérard, CEA-LIST/LISE, France
- Susanne Graf, Univ. Joseph Fourier-CNRS-VERIMAG, France
- Ileana Ober, University of Toulouse-IRIT, France
- Iulian Ober, University of Toulouse-IRIT, France (*chair*)
- Dorina Petriu, Carleton University, Canada
- Andreas Prinz, University of Agder, Norway
- Bernhard Rumpe, RWTH Aachen, Germany
- Bran Selic, Malina Software, Canada
- Martin Törngren, KTH Stockholm, Sweden
- Tullio Vardanega, University of Padua, Italy
- Eugenio Villar, Universidad de Cantabria, Spain
- Thomas Weigert, Missouri S&T, USA
- Tim Weilkiens, OOSE Innovative Informatik GmbH, Germany

Workshop program

9:50-10:30 Opening session

1. *Welcome address*
2. *Tool-Supported Model-Driven Validation Process for System Architectures*
   Andre Pflueger, Wolfgang Golubski and Stefan Queins

10:30-11:00 Break

11:00-12:30 Session 1 : System architecture modeling and analysis

1. *HiLeS-T: an ADL for early requirement verification of Embedded Systems*
   Horacio Hoyos, Rubby Casallas and Fernando Jimenez
2. *Functional Validation of AADL Models via Model Transformation to SystemC with ATL*
   Pierre Bomel, Dominique Blouin, Mickael Lanoe and Eric Senn
3. *Automatic SysML-based Safety Analysis*
   Philipp Helle

12:30-14:00 Lunch

14:00-15:30 Session 2 : From models to implementations

1. *Real–Time Design Models to RTOS-Specific Models Refinement Verification*
   Rania Mzid, Chokri Mraidha, Jean-Philippe Babau and Mohamed Abid
2. *Component-Based Models for Runtime Control and Monitoring of Embedded Systems*
   Tobias Schwalb, Tobias Gädeke, Johannes Schmid and Klaus Müller-Glaser
3. *Model-based Development of Embedded Systems' User Interfaces*
   Jelena Barth, Bernd Westphal and Stephan Arlt

15:30-16:00 Break

16:00-17:30 Session 3 : Model semantics and synthesis

1. *Formal Execution Semantics for Asynchronous Constructs of AADL*
   Jiale Zhou, Andreas Johnsen and Kristina Lundqvist
2. *Automatic synthesis from UML/MARTE models using channel semantics*
   Pablo Peñil, Héctor Posadas, Alejandro Nicolás and Eugenio Villar
3. *Automatic Transformation of Abstract AUTOSAR Architectures to Timed Automata*
   Stefan Neumann, Norman Kluge and Sebastian Wätzoldt

17:30 Workshop Ends

Contents

Tool-Supported Model-Driven Validation Process for System Architectures

André Pflüger
Westsächsische Hochschule
Dr.-Friedrichs-Ring 2a
08056 Zwickau, Germany
andpfl@fh-zwickau.de

Wolfgang Golubski
Westsächsische Hochschule
Dr.-Friedrichs-Ring 2a
08056 Zwickau, Germany
golubski@fh-zwickau.de

Stefan Queins
SOPHIST GmbH
Vordere Cramergasse 13
90478 Nürnberg, Germany
stefan.queins@sophist.de

ABSTRACT

Designing system architecture is still an error-prone process and a great challenge. The development of complex embedded systems like radar systems is very cost-intensive. Therefore it is important that system architects are supported by appropriate tools. Our UML-based process focuses on validating the architecture against system requirements and analyzing the impacts of requirement or architectural changes. In this paper we present a supporting tool providing automatization possibilities for the validation process. This is a major breakthrough as it reduces the need for repetitive, time consuming and mindless validation process to be conducted manually. The tool is able to handle all the requirements, including the requirements' interconnections with one another, and increase process usability.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation—*system architecture, requirements, simulation*

General Terms

Design, Verification

Keywords

Validation, system architecture, simulation, model-driven, UML

1. INTRODUCTION

The general technological progress allows for the development of embedded systems with increasing complexity. This involves a rising number of requirements including their dependencies among each other. Therefore the challenges of designing a good system architecture are to identify the relevant requirements and to analyze the impact of changing the requirements on the architecture. Every time requirements are changing the architect has to revalidate the architecture. This time-consuming task should be supported by convenient tools that reduce the need for the process to be completed manually.

In this paper we describe our proposed process [1] [2] and present a tool that helps the architect improve system development. Our approach defines a model-driven validation process based on the Unified Modeling Language (UML) and simulations as validation technique. By defining validation targets, architecture-specific aspects, assigned with architecture relevant requirements a validation specific view is created. It enables the architect to handle all requirements with their interconnections and to analyze the impact of requirement and architecture changes. For every validation target the architect configures an examination simulation to validate the target against the assigned requirements. The architecture is only valid if all validation targets are valid. After modifications of the architecture or changes to the requirements the architect must rerun the validation process. The process provides automation possibilities to minimize this effort. It improves the quality of the requirement and architecture change management and of the system design process leading to a higher quality of the system architecture.

Section 2 describes briefly the validation process, which is applied to an example from the area of embedded systems in section 3. The support tool is described in section 4. Section 5 describes our experience with the approach and its benefits. Section 6 presents related work for the approach and the support tool before section 7 concludes the paper.

2. VALIDATION PROCESS

There are many definitions of the terms validation and verification in literature [3] [4] [5] [6] [7] [8]. For our work we generally define validation as a process performed during or at the end of a development phase to check the resulting artifact(s) against the corresponding requirements specified prior to the phase. Verification is a process performed for approval of system parts or the entire system. The main differences are the checked artifacts and the point of time in system engineering at which the process is applied. Figure 1 illustrates the process by an UML activity diagram. In the first step the architect has to identify the architecture relevant requirements to determine the validation targets. A validation target sums up all requirements of one architecture-specific aspect and does not possess comparative values. All together they create an abstract layer suitable for architecture validation. Each target is connected to

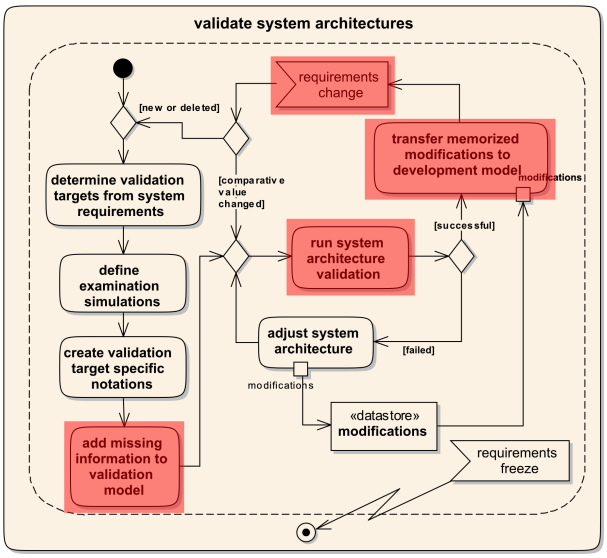

Figure 1: Process for system architecture validation

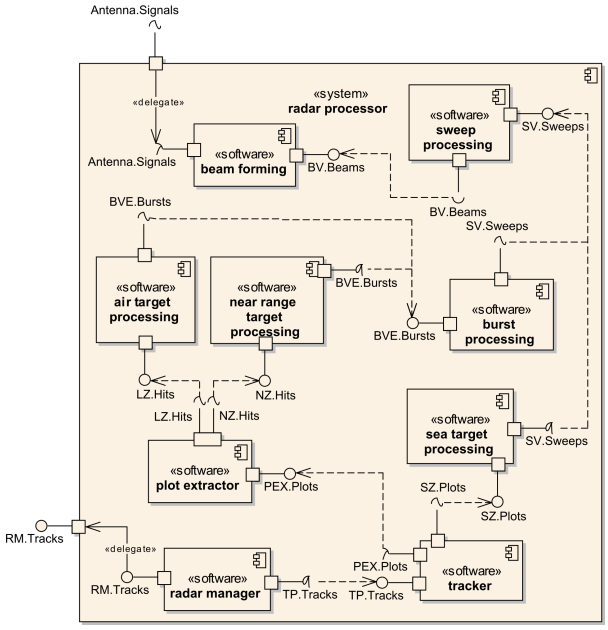

Figure 2: Software view of a simplified radar processor in development model

at least one architecture-relevant requirement whereas one requirement can be associated to several targets. In the next step the architect has to define examination simulations for each validation target. The examination simulation is a procedure to check if the architecture matches the corresponding requirements. The data source for these simulations is the validation model which contains only validation relevant data unlike the development model containing all development relevant data. The validation model provides a view concealing all non essential architecture validation relevant data from the architect. Knowing the examination simulation the architect can define the validation target specific notation to define how the input data for the simulation is added to the validation model. For creating the notation our approach uses the UML profile mechanism. The architecture is entirely validated by running all examination simulations and comparing the results with the comparative test values of the implicit associated requirements (see action *run system architecture validation*). The system architecture validation fails when the assigned simulation cannot validate one or more validation target. In this case the architect must adjust the architecture in the validation model and rerun the architecture validation. The system architecture validation succeeds only if all validation targets are valid. Architecture modifications relevant for the development model are transformed from the validation model. The process has to be restarted if new requirements are gathered or existing ones are deleted. If comparative values are changed, the architect can skip the first steps and must only rerun the architectute validation. The process terminates with the requirements freeze. Detailed information can be found in [1] and [2].

3. EXAMPLE OF USE

The example in this paper is reused from a previous experiment [2] but extended for the purpose of demonstrating the support tool that is not described in the previous work [2]. We also use an improved simulation technique. The example is from the area of embedded systems: a simplified radar system. It receives echoes of electro-magnetic signals from an antenna, processes these data in real-time and provides visible tracks on a radar display. Figure 2 illustrates the software view of the radar processor in the development model containing nine software components. It shows that in a radar processor the data processing is almost sequential. It is only separated by data processing variants due to different types of signals which are in our example: sea targets, air targets and near range targets. These signals are characterized by different electro-magnetic wave forms and processing algorithms allowing the radar engineer to detect objects in a short or far away distance with different resolutions. E.g. objects on surface of the water are usually not very fast in comparison with missiles in the air and therefore they have to be treated differently. The overall goal is to filter out signals reflected by objects in radar range which the radar engineer does not want to see on his radar display. These tracks depend on the field of usage; e.g. air surveillance radars at airports need to detect not only planes but also groups of birds because they are a threat to airplane engines. The system processes a continuous stream of data so that the computing performance for data of a certain point in time is influenced by the processing of past and future signals. In the development model each software component (UML component with stereotype *software*) has to have a dependency connection with stereotype *executedOn* to a hardware component indicating its execution environment. In our example there are three hardware boards with two processing units, one microprocessor and one Field Programmable Gate Array (FPGA), each. The software component interfaces are mapped to the physical ones represented by ports on UML components with stereotype *hardware* allowing the architect to trace data flow on the hardware and to overview dependencies between the boards.

According to the first step in the validation process (see Figure 1) the architect detects validation targets from system requirements. For that reason he identifies the following six

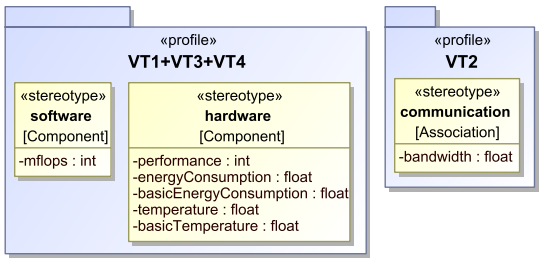

Figure 3: Validation target specific notations

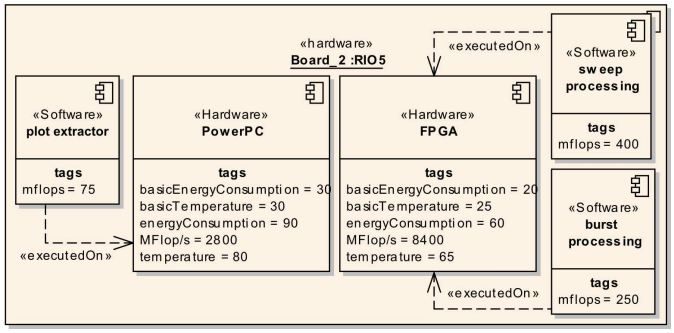

Figure 4: Extract of the validation model

architecture relevant requirements:

- The system shall provide tracks for fed in test data after 320ms. (A1)
- The system shall provide plots for fed in test data after 280ms. (A2)
- The system shall be able to compute 3500 MiBit/s data received from antenna. (A3)
- The system shall consume less than 220W. (A4)
- The temperature of each FPGA shall be less than 50°C. (A5)
- The temperature of each microprocessor shall be less than 70°C. (A6)

The first three non-functional requirements can be assigned to two validation targets (VT). A1 and A2 deal with the processing time of the system (VT1), A3 with the communication infrastructure (VT2). A4 can be assigned to the aspect energy consumption (VT3) and A5 and A6 to hardware temperature (VT4). The next steps are to define the examination simulation and the validation target specific notation for each target. Figure 3 shows the UML profiles used for validation target specific notations. VT1, VT3 and VT4 are combined in one profile because VT3 and VT4 depend on the examination simulation of VT1. The processing time to provide tracks and plots is calculated by required floating point operations of the software components. That is why the stereotype software has the tagged value *mflops* specifying needed floating point operations for the software component. The stereotype *hardware* has the tagged value *performance* providing the performance of processing units in MFlop/s. In order to calculate system's processing time we have to consider the amount of parallel processed software components on the same processing unit influencing the performance available for assigned software components. If we assume linear energy consumption and linear temperature rising for processing units in combination with a basic value, these data can be calculated by using processing unit load deduced by VT1's examination simulation. We add the tagged values *energyConsumption* and *basicEnergyConsumption* to the stereotype *hardware* as notation for VT3 and *temperature* and *basicTemperature* for VT4 (see Figure 3). For VT2's examination simulation the bandwidth of the communication device from antenna to radar system is required. We add the stereotype *communication* with tagged value *bandwidth* to the association representing physical link between the board and the hardware component executing the first software component in the processing chain.

After the determination of the validation targets, their corresponding examination simulations and notations, the architect creates the validation model using the support tool described in section 4.2. The tool loads the development model, lists all available elements according to the model tree and enables the architect to select validation relevant structural and behavioural elements and to transform them into the validation model. After transformation the architect can add missing validation-specific data (e.g. amount of flops required by software components) by using the UML profiles. Figure 4 shows an extract of the resulting validation model. The results of the system architecture validation, i.e. of the four validation targets, are shown in table 1 to 3. The system architecture passes the validation and therefore fulfills the regarded system requirements. There has been no architecture modification in the validation model so that synchronization of the validation and development model is not necessary. In the course of system development requirements are changing. In our example the customer changes requirement A1: Tracks shall be provided after 310ms instead of 320ms. This modification causes no change to the validation targets itself because they are defined independently from specific values. According to the validation process (see Figure 1) the next step is *run system architecture validation*. It fails because VT1 cannot be validated: 313ms is greater than 310ms. The architect has to change the architecture in order to fulfill the new requirement. He tries to increase performance by assigning the software component *sweep processing* to the PowerPC of Board_2. This modification is memorized because it has to be transferred to the development model in case of a successful validation. Although VT1 is valid after this architecture modification, the validation fails due to energy consumption. The support tool provides an overview of the validation status by a color-coded list of the validation targets for the architect (see figure 6). Each target is colored accoording to its status whereas green means valid and red means invalid. The detailed results of the simulation are shown in table 4 and 5. The tool processes data for graphical presentation to support the architect in analyzing simulation results. Unlike formal techniques the dynamic model-driven simulation provides not only final but also interim values. Those support the architect in analyzing failed validations. An example is the presentation of the processor load according to the

Table 1: Result First Simulation VT1

Data	Processing time [ms]
Tracks	313
Plots	277

Table 2: Result First Simulation VT2

Communication line	Bandwidth [Mibit/s]
Antenna - radar processor	5200

simulation time in an coordinate system. By analyzing the data of all performed examination simulations the architect is able to identify the problem. The processing unit load of the PowerPC on Board_2 increased due to raising energy consumption which cannot be compensated by decreasing energy consumption of the FPGA on Board_2. The architect restores the former architecture in the validation model, exchanges the software components *burst processing* and *near range target processing* and reruns architecture validation. The validation for each target succeeds resulting in transferring all modifications to the development model. Thus the two models are synchronized.

4. SUPPORT TOOL

The four actions highlighted in figure 1 indicate automation and support possibilities of the process provided by our support tool. Figure 5 shows the five use cases available for the architect. Section 4.1 goes into detail for *manage validation targets*, *support impact analysis* and *manage requirements*. Section 4.2 addresses model-to-model transformations and section 4.3 focuses on a model-driven examination simulation for heavily dependent validation targets.

4.1 Validation Target Management

Applications like Rational DOORS are capable of managing requirements. By adding attributes it could be possible to manage validation targets in some manner. However, the possibilities to directly start the assigned examination simulation, to compare the examination simulation results with the comparative values, to persist and restore validation results and to evaluate requirements and validation target dependencies are not available in such tools. For these reasons we developed our own managing tool for validation targets enabling the architect to handle the targets and the assigned requirements. The architect can import requirements from an external *Requirements Management Tool* by a XML or

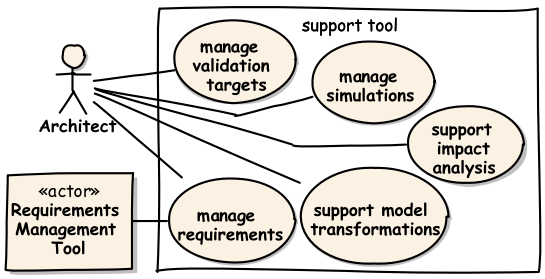

Figure 5: Use case diagram of the support tool

Table 3: Result First Simulation VT3 and VT4

Processing Unit	Processor load [%]	Temperature [°C]	Energy [W]
B1_FPGA	86	47	34
B1_PowerPC	8	52	38
B2_FPGA	84	45	33
B2_PowerPC	8	52	38
B3_FPGA	17	38	26
B3_PowerPC	23	67	48
Total energy consumption			**217**

Table 4: Result Second Simulation VT1

Data	Processing time [ms]
Tracks	306
Plots	270

spread sheet interface into an empty or existing set of requirements and validation targets. If existing requirements are changed by importing a requirements file, the requirements and the assigned targets are highlighted. The architect can manage validation targets including the assignments to the architecture-specific requirements. For the validation targets he has to choose comparative values and examination simulations. The architect can start the validation of the entire architecture or even for single targets and present the results in a graphical overview (see figure 6). Several filters enable the architect to handle the amount of requirements and their interconnections. Requirements without an association to a validation target can be filtered and displayed. It is possible to show all assigned requirements of a selected validation target as well as all assigned validation targets of one selected requirement. Furthermore the tool lists all indirect connected validation targets by analyzing the assigned requirements and evaluating their connections to other requirements. If those requirements are assigned to another validation target than the selected one it is displayed by the tool including the connecting requirements. This view supports the architect to identify complex dependencies between validation targets and helps to realize the impact of certain requirements on the system architecture.

4.2 Model Transformation Support

The support tool does not only manage the files of the development and validation models but also provides model synchronization and transformation of elements between them. Development and validation information are separated in our validation process to avoid information overkill in the development model and to enable the disengagement of data in the validation model from data in the development model. Thus, the validation relevant data is independent from changes in the development model caused by development progress. These advantages come at the price of extra effort for creating the validation model, for synchronizing the data and for checking consistency. The effort can be reduced by using the meta model based modeling language UML for documentation. If development and validation model are based on the same meta model, modeled data can be transformed from one model to an existing or new one without great effort. Modeled data can be checked against self-defined rules to ensure consistency and it can be synchronized by analyzing

Table 5: Result Second Simulation VT3 and VT4

Processing Unit	Processor load [%]	Temperature [°C]	Energy [W]
B1_FPGA	86	47	34
B1_PowerPC	8	52	38
B2_FPGA	66	40	30
B2_PowerPC	28	69	52
B3_FPGA	15	36	25
B3_PowerPC	21	60	47
Total energy consumption			**226**

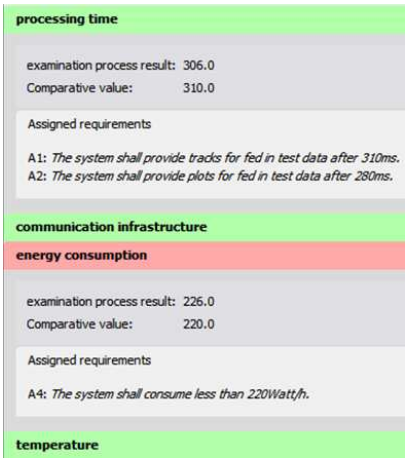

Figure 6: Color-coded validation target overview according to validation status

and comparing model content.

4.3 Simulation Management

The support tool manages the examination simulations of the validation targets and provides an interface for reading data from the validation model. This data can be utilized by the simulations as data source for their algorithms on the one hand and to configure the simulation, e.g. the hardware infrastructure or the software-hardware mapping, on the other hand. The simulations provide their results over an universal result interface whereby the tool updates the validation status of the targets and the whole architecture accordingly. These interfaces enable the architect to use simulations suitable for the level of detail in the current development status and to modify the architecture without reprogramming the simulations. For the radar system we developed a simulation considering multiple validation targets. Although the level of detail is not high, even domain experts cannot foresee the simulation results because there are to many influencing factors.

5. BENEFITS AND EXPERIENCES

Our approach has been developed with the help of projects of the SOPHIST GmbH[1] and our experiences from developments of embedded systems. One of the projects has to deal with about 40 processing units and 70 software components. The used examination simulations are generally similar to the introduced example. Our experience shows that extra work caused by this validation process amortizes within few iterations of system architecture design and requirement changes. According to our field trials the approach can be learned in a one day workshop for experienced system engineers, beginners require two days. The benefits of our approach can be outlined as followed:

- It can be integrated into iterative architecture design improving the quality of the process.
- The required time and effort for system architecture validation can be reduced by automation of examination simulations and of validation process' steps.
- The validation targets create an architecture-specific view allowing the architect to keep track of all relevant requirements including their dependencies, direct and indirect.
- The architect can analyze the impacts of requirement changes on the architecture via their connections to the validation targets.
- The validation process also indicates secondary effects of architecture changes.
- Separation of validation and development data allows independent working on both models.
- UML profile mechanism enables the architect to add arbitrary validation-specific data without much effort and independently from other notations.

6. RELATED WORK

The elicitation of requirements in particular non-functional requirements is not part of our approach. Nevertheless, non-functional requirements are considerable for architecture design and therefore we require a preferably complete set of non-functional requirements satisfying basic quality properties like testability or unambiguousness. An example for a suitable approach is described in [9]. This systematic elicitation process allows finding inconsistent and conflicting requirements based on a requirements meta model. This model could also be supportive for indentifying architecture relevant requirements to determine validation targets and examination simulations.

According to [10] the simulations used in our approach can be classified as dynamic validation technique in contrast to formal techniques like model checking or theorem proofing. The simulations are evaluating the system behaviour based on data modeled in UML. This semi-formal modeling language does not have the accuracy of the mathematical models normally used by formal verification techniques. [11] describes a system design verification approach using the object-oriented equation-based modeling language Modelica. The data required for verification is added to an UML model by the ModelicaML UML profile. The formalized requirements are observed by violation monitors, roughly comparable to the validation targets. The main target of the approach is the detection of design errors whereas our approach answers the question: Could it be possible to satisfy the requirements? Similar approaches due to the main target are e.g. [13] dealing with hard real-time systems, [12]

[1] `http://www.sophist.de/en/start`

using modeled data for system testing and [14] providing an development approach for the automotive domain. The focus of the latest is more on software than on system architectures using AUTomotive Open System ARchitecture.

Adding information to the model is done by using default UML elements and elements provided by domain specific profiles like MARTE, SysML, AADL or EAST-ADL. However, these profiles do neither provide a process for architecture validation nor allow adding arbitrary data. Therefore, our approach uses the UML profile mechanism to create validation specific profiles to add data which cannot be added by existing profiles.

Scenario-based analysis methods for software architectures like SAAM or ATAM [15] can also be applied on system architectures[2]. Nevertheless, they are evaluating a set of architectures according to chosen quality properties to select the one that fits best. Our approach focuses on finding valid architectures which could be analyzed by those methods afterwards. There is a validation and verification component for the tool Simulink[3]. It requires detailed information about the developed system which is not necessarily available at system design level. Although it would be possible to validate the entire architecture even in an early development phase using the harness model, the main focus of Simulink is on developing single parts, e.g. FPGAs, of the system. Nevertheless, Simulink does not use UML for modeling and the mentioned component is restricted to functional and security requirements.

The support tool is highly specific to the validation process. It provides a model-driven simulation to deal with heavily dependent validation targets. [16] describes an approach to distribute simulation code to different processing units by changing the virtual machine code leaving the simulation code unmodified. This could be used for a radar simulation based on a detailed system architecture. However, there is no radar simulation available which can be configured by an UML model and adjusted to different level of details.

7. CONCLUSION AND FUTURE WORK

In this paper we presented a tool supporting the architect in validating the system architecture against system requirements. It provides data management, model transformation and simulations reducing the validation effort and enabling the architect to analyze the impact of requirement and architecture changes on the validation status of the system architecture. Currently our approach and the supporting tool are applied to another domain, the domain of business processes. First results are promising.

8. ACKNOWLEDGMENTS

We give thanks to Tommy Hartmann, Christian Poßögel and Samuel Weigelt for implementing the support tool.

9. REFERENCES

[1] Pflüger, A., Golubski, W., Queins, S.: Validation of System Architectures against Requirements. In: 2010 International Conference on Systems, Computing Sciences and Software Engineering, part of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE). Springer (2012)

[2] Pflüger, A., Golubski, W., Queins, S.: Model Driven Validation of System Architecture. In: 13th IEEE International Symposium on High-Assurance Systems Engineering (HASE), pp. 25-28. IEEE Computer Society (2011)

[3] Boehm, B.W.: Software Engineering Economics. Prentice Hall (1981)

[4] Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering. Addison-Wesley Longman (2003)

[5] Grady, J.O.: System Validation and Verification. CRC Press, Boca Raton (1997)

[6] International Council on System Engineering: INCOSE Systems Engineering Handbook (version 3.2.2), http://www.incose.org

[7] Pohl, K.: Requirements Engineering - Fundamentals, Principles, and Techniques. Springer (2010)

[8] Navy Modeling and Simulation Management Office: Modeling and Simulation Verification, Validation and Accreditation Implementation Handbook. Technical Report, Department of the Navy, USA (2004)

[9] Dörr, J.: Elicitation of a Complete Set of Non-Functional Requirements. Doctoral Thesis. University of Kaiserslautern. Fraunhofer Verlag (2010)

[10] Debabbi, M., Hassaine, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering - Assessing UML/SysML Design Models. Springer (2010)

[11] Schamai, W., Helle, P., Fritzson, P.,Paredis, C.J.J.: Virtual Verification of System Designs against System Requirements. In: Models in Software Engineering - Workshops and Symposia at MODELS 2010. LNCS, vol. 6627, pp. 75-89. Springer (2011)

[12] Aydal, E.G., Paige, R.F., Utting, M., Woodcock, J.: Putting Formal Specifications under the Magnifying Glass: Model-based Testing for Validation. In: Second International Conference on Software Testing, Verification and Validation (ICST), pp. 131-140. IEEE Computer Society (2009)

[13] Hall, R.J.: Validating Real Time Specifications using Real Time Event Queue Modeling. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 79-88. IEEE (2008)

[14] Holtmann, J., Meyer, J., Meyer, M.: A Seamless Model-Based Development Process for Automotive Systems. In: Software Engineering (SE) 2010 - Workshopband, LNI volume P-184, pp. 79-88, Bonner Köllen Verlag (2011)

[15] Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Longman (2001)

[16] Gray, I., Audsley, N.C.: Targeting Complex Embedded Architectures by Combining the Multicore Communications API (MCAPI) with Compile-Time Virtualisation. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp.51-60. ACM (2011)

[2] http://www.sei.cmu.edu/architecture/tools/evaluate/systematam.cfm

[3] http://www.mathworks.com/verification-validation/

HiLeS-T: an ADL for early requirement verification of Embedded Systems

Horacio Hoyos Rodríguez
Universidad de los Andes
Cra 1 N° 18A- 12
Bogotá, Colombia
h.hoyos95@uniandes.edu.co

Rubby Casallas
Universidad de los Andes
Cra 1 N° 18A- 12
Bogotá, Colombia
rcasalla@uniandes.edu.co

Fernando Jiménez
Universidad de los Andes
Cra 1 N° 18A- 12
Bogotá, Colombia
fjimenez@uniandes.edu.co

ABSTRACT

Verification of functional and non-functional requirements throughout the design process is a cost-effective solution when compared to a build-test validation process. By using a model based design process and by describing system behavior with a formal model, model checking becomes a viable solution to perform requirement verification at early stages of the design process. This paper presents how the HiLeS ADL can be used to express the behavior of the system with a Petri Net and how to use that representation to perform system verification. HiLeS is used as a intermediate stage of a model driven automated virtual prototype design framework, in which SysML is used for capturing requirements and system modeling.

Categories and Subject Descriptors

B.6.2 [**R**]: eliability and testing; B.6.3 [**D**]: esign Aids; D.2.2 [**D**]: esign Tools and Techniques

General Terms

ADL; Model Driven Technichs, Verification

1. INTRODUCTION

Embedded system design is a challenging domain, in which a set of requirements must be transformed in a detailed specification that can be used for manufacturing. Many works are trying to find a cost effective way to do the task, with solutions that range from proposing languages for system modeling [13], methodologies that define the necessary steps of the design process [18], all the way through tools that automate the design steps (usually transformations from abstract descriptions to more detailed ones) [16].

However, no matter which of the approaches is taken, the cost of manufacturing an embedded system is too high to perform a build-test validation process. To overcome this, verification of the design with respect to the functional and non-functional requirements should be carried out through the design process [5] and ideally as early as possible where it is more cost effective.

According to the embedded design scenario presented by Gajski [7], there are four phases each one identified by an abstraction layer of the Y-Chart: system, processor, logic and circuit. In each of the phases a particular language can be used to describe the system at the corresponding level. At lower levels of abstraction, Virtual prototypes[1] can be used to validate the system requirements by simulation. At higher levels of abstraction (i.e., system and processor), system components are described with nondeterminate specifications that give constraints on the behavior, but not the details of the behavior of the simplest components themselves. These levels of abstraction and the simplification they provide help model checking and formal verification at an early design phase. Further, if the description language used is based on a precise mathematical model, or a Model of Computation (MoC), verification can be done with tools of guaranteed performance.

This paper presents an extension to the work presented in [14] with emphasis in the support for early design formal verification. In [14] SysML is used as the language to describe the embedded system at the system level and HiLeS as the Architecture Description Language to describe the system at the processor level. We use a limited set of SysML diagrams and diagram elements to build the system level model based on the EIA-632 Systems engineering process [15]. The SysML model is used to capture requirements and to define the *Logical Solution* of the system. The logical solution describes the structural and behavioral aspects of the system. The HiLeS [11] formalism and its tooling allow designers and engineers to produce a high level design defining the functional decomposition, hierarchy and structure of the system. HiLeS proposal is based on a global system design methodology and a formal behavioral model based on Petri Nets. In our work, in order to improve the design process of embedded systems, from the system to the logic levels, we seek to provide tools to automate the synthesis process[2] and to support the validation and verification at

[1] Graham Hellestrand, How virtual prototypes aid SoC hardware design, `http://www.embedded.com/columns/technicalinsights/20300463`

[2] *synthesis*, as proposed by Gajski [7], is the process of converting a model of the system at a given abstraction layer

the different abstraction levels.

Since the HiLeS formalism is based on Petri Nets the processor model can be used for formal verification and model checking. Further, since we generate automatically the HiLeS-T model from the SysML model, using model driven techniques, the verification process can be done at an early design phase. In our work we focus on soft real-time only hardware systems. In the case of soft real time systems, timing requirements specify the average time that system response to events must be kept[3]. Since we are interested in timing aspects, we have extended HiLeS to HiLeS-T where the MoC is based on Timed Petri Nets (TPN). Activity constraints are used in the SysML model to express timing requirements as duration constraints on activities. As part of our work, we have extended HiLeS into HiLeS-T to:

1. Facilitate the synthesis from SysML to HiLeS-T. Although HiLeS behavior representation allows us to precise the semantics of activity diagrams new concepts where needed to support some of the activity diagram elements, more precisely interruptible regions.

2. Support ordinary Timed Petri Net as part of the HiLeS-T behavior representation (or HiLeS Control Net), so behavior verification (liveness, deadlocks, etc.) could be performed taking time requirements into account.

3. Facilitate the synthesis from HiLeS-T to Hardware Description Languages. We added support for vector ports and specialized data flow connectors to math more easily the connector types supported by VHDL-AMS and Verilog-AMS.

This paper presents how HiLeS-T is used to represent system behavior, how to perform behavior verification and how time requirements can be added to the model to further allow time-dependent verification. Section 3 how to use SysML for system level modeling and time requirement specification and Section 4 presents HiLeS-T. Section 5 presents the verification process. Section 6 presents related work and section 7 concludes.

2. OVERVIEW

Testing is still the primary technique for validation and its cost can be as high as 50% to 70% of total development [17]. As shorter times to market are demanded by market trends, it is no longer economically viable to wait for building the System-on a Chip (SoC) to carry out all the validation processes. By using a model based design process, model checking becomes a viable solution to perform verification and validation at early stages of the design process. In order to provide model checking of concurrent systems with constraints on time, Timed Petri Nets have been proposed as an alternative to timed automata [2]. An important interest of TPNs, and PNs in general, lies in their applicability to the verification of boundedness, coverability, reachability, and other properties in infinite-state systems. In a TPN, a transition can be fired if it is enabled (every input place contains the required number of tokens) and if the time since it has been enabled lies in the specified firing interval.

As mentioned previously the HiLeS-T formalism has a behavior model based on TPN, known as a HiLeS Control Net (HCN). The HCN serves two purposes: i) express the behavior of the system as a sequence of component executions (parallel execution is allowed); ii) Capture the time execution information of components (in the transition's firing interval). In the case of system verification, the time execution information is the desired execution time of the component, i.e., the time requirements. In order to do model checking, we have to extract from the HCN the TPN for what we use a model transformation.

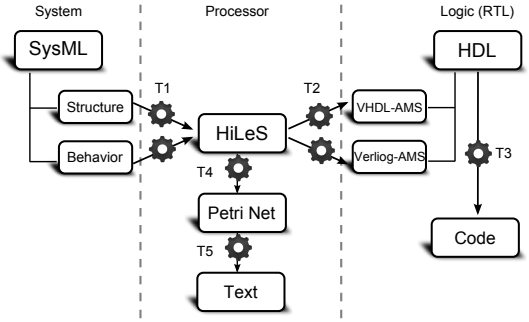

Figure 1: HiLeS Model transformation chain

In our approach we use model driven techniques (MDT) to provide automatic synthesis from the system level model to the logic level model. To do so we have built a model transformation chain (MTC) that covers three of the domains of a design process: system, processor and logic. At each of these levels we have chosen a specific language for modeling: SysML for system, HiLeS for processor and VHDL-AMS or Verilog-AMS for logic. To provide verification capabilities at the processor level the MTC also covers the Timed Petri Nets domain.

Figure 1 presents an overview of the MTC. Model transformations T_1 and T_2 provide the synthesis from system to processor and from processor to logic levels respectively. T_3 is a model to text transformation that generates either VHDL or Verilog code. T_4 is a transformation from the HiLeS model to a TPN model (TPN extraction) and T_5 is a model to text transformation to represent the TPN in a textual format understandable by the PN analysis software. In our case we use TINA [1] as the model checking tool for PN analysis. (pegamento)

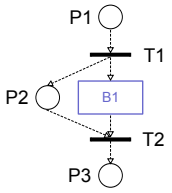

Figure 2: HiLeS Control Net

Figure 2 presents what can be considered the building block

to a lower (more detailed) one.

[3]Douglass, B. P. 2001. Capturing real-time requirements. `http://www.embedded.com/9900356`

of the HiLeS Control Net and represents the interaction between the Time Petri Net elements of the HCN and a system component. System components are represented by blocks in the HCN. When transition T1 fires block B1 is signaled to start execution. Execution time of B1 is associated to transition T2. Since the place P2 has a token, transition T2 is enabled but will no fire until some time into the firing interval has elapsed.

3. SYSML MODELING & CASE STUDY

In [14] we introduced a simple tank level control for a sugar cane production system, as presented in figure 3. The system under design (SUD) consists of a level control for the tank, which signals the evaporator when the tank is full and the vacuum pans when the tank is empty. The global requirements of the system can be expressed in natural language as follows:

> The tank level must be calculated based on the tank's input and output flows. A *full* alarm is generated if the tank is at or over 95% capacity and an *empty* alarm is generated if the tank level is bellow 5%. Both alarms must be generated with a maximum delay of 100ms after the respective level has been reached. The full and empty alarms must signals the *Evaporator system* and the *Vacuum Pan system* respectively. Flow and alarm signals are 4-20 mA.

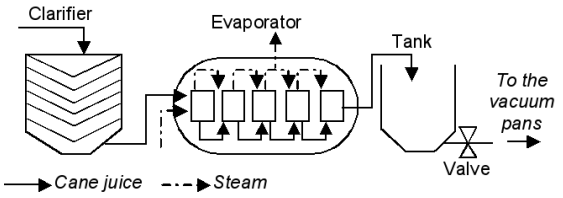

Figure 3: Sugar cane process

After a requirement analysis [10, 4] the design engineer can then expand this requirements to derive the system technical requirements and use SysML requirement diagrams to capture them. Figure 4 presents a (partial in order to limit the scope of the paper) SysML requirement diagram with the functional requirements of the system, derived from the stakeholder's requirements. The stakeholder requirements are placed outside of the *Functional* package. Table 1 presents details of the system requirements which we will use during the following sections as part of our case study.

After the requirements have been captured, the designer builds the Logical Solution, which is used to define the structural and behavioral features of the SUD. In the structural part, the system is broken down into hierarchical components. Figure 5 presents a partial view of the system's block definition diagram (bdd). It can be seen that the system has a *Flow Processor* component which is responsible for the level calculation and alarm generation. This component is composed of the *Level Calculator* and *Action* components. The Level Calculator is responsible for performing the analog to digital conversion (req `SF:0001`) and calculating the tank level (req `SF:0002`). The Action component

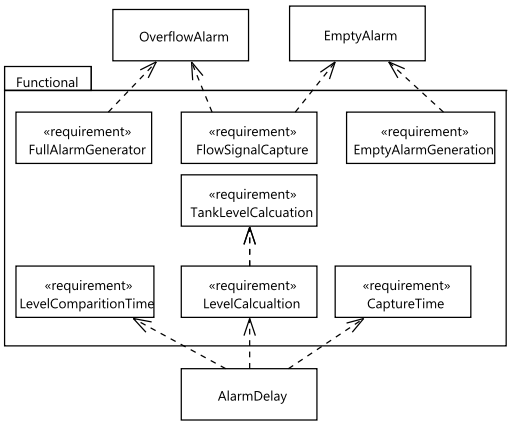

Figure 4: Sugar cane system functional requirements

is composed of a *emptyAlarm* and an *overflowAlarm* (both of which are Alarm Generator Components). These components are responsible for comparing the tank level with the control levels and generating the respective alarm (req `SF:0003` and `SF:0004`).

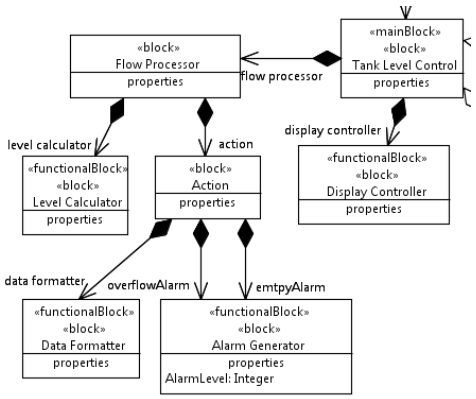

Figure 5: Sugar cane block definition diagram

3.1 Activity diagrams & Time requirements

The bdd is good for defining what components are need to provide the required functionality but it does not provide any behavior information and hence it can not naturally be used to represent time requirements. Different diagram are available if SysML for behavior representation. In [14] we used sequence diagrams, which how now been replaced by activity diagrams for two reasons: i) we believe that for systems that are only hardware an *action invocation* semantics suits better than an message send/receive one; ii) activity diagrams are based on PNs and synthesis of the behavior from system to processor level is easier.

In order to express time requirements in activity diagrams we use *Local Postcondition Duration Constraints*. The first step is to associate the time requirements to a component in the system. This can be easily done by analyzing the functional requirements and identifying to which of the proposed components does the requirement affect. The use of activity

Table 1: Partial SUD Requirements

Id	Name	Text
S01:001	OverflowAlarm	A full alarm is generated if the tank is at or over 95% of its level
S01:002	EmptyAlarm	An empty alarm is generated if the tank level reaches or is bellow 5%
S01:003	SignalInterface	Flow and alarm signals are 4-20 mA
S01:004	AlarmDelay	Alarms must be generated with a maximum delay of 100ms after the respective level has been reached
SF:0001	FlowSignalCapture	ADC with 12bit resolution for flow signal capture
SF:0002	TankLevelCalculation	Tank level must be calculated from input and output flows
SF:0003	FullAlarmGenerator	Generate an alarm signal when level >= 95%
SF:0004	EmptyAlarmGenerator	Generate an alarm signal when level <= 5%
SF:0005	LevelCalcuationTime	Calculation time < 50ms
SF:0006	LevelComparitionTime	Comparition time < 10ms
SF:0007	CaptureTime	Flow capture time < 5ms

partitions (also known as swim-lanes) in activity diagrams is one of the solutions for linking behavior to structure [6]. In our approach we use *Call Behavior Actions* to represent the invocation of the functionality of a component and with the use of activity partitions we can specify what component is responsible for executing the action. Figure 6 presents the activity digram of the action component. An activity partition is used to specify that the *Format* action is executed by the *formatter* component and the *Alarm* action is executed by the *empty* and *overflow* component. Requirement `SF:0006` states that the time to compare the level and generate an alarm should be less than 10ms. We have added a duration constraint to the decision node and the alarm activity to capture this information. The duration constraint is represented with a note attached to the node with the *Local Postcondition Duration Constraint* tittle, and the constraint expressed as an integer. In this case we have selected 3ms as the constraint to keep the execution below the 10ms.

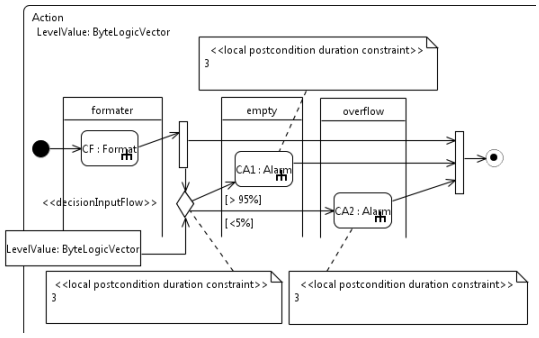

Figure 6: Flow processor component activity diagram.

4. HILES-T & TPN

The HiLeS-T formalism provides a formal hierarchical structural description of a system and a precise semantics to represent its behavior based on Timed Petri Nets. HiLeS supports asynchronous and concurrent behaviors and mixed representations with event driven dynamics. This characteristic allows modeling of heterogeneous systems (most modern embedded systems are heterogeneous in nature), i.e. they include multi-domain physics and multidisciplinary domains (digital and analog electronics, software, RF, optics, mechanics, and so on).

HiLeS-T semantics is an extension to Timed Petri Nets and defines the interaction between the architectural components of the system having into account that the components have a finite duration. The extension consists in considering system components, represented by structural or functional blocks, as part of the PN. Specifically they act as places, meaning that arcs can be connected from transitions to blocks and from blocks to transitions. From an execution point of view, when a transition fires it places a token on the blocks connected to its output arcs and for a transition to fire, in all places and blocks connected to its input arcs tokens must exist.

To perform model checking with PN analysis tools, the TPN must be extracted from the HiLeS-T model in order to remove the blocks and leave a "pure" TPN. Since it is required that the TPN holds the time requirements information we defined a method for translating duration constraints into the HiLeS-T model so that they would be maintained during the TPN extraction. The execution of the HiLeS-T Control Net (HCN) presented in figure 2 is as follows: when T1 fires, it places a token in block B1. According to the HCN semantics, B1 will "hide"the token from T2 till it completes executing. After B1 finishes executing, it shows the token to T2 which can be fired because it is enabled. The same behavior could be represented with a TPN if the execution time of B1 is associated to transition T2 i.e., by associating the execution time of B1, t_E to both limits of the firing interval of T2 and removing B1 (TPN extraction). From an execution point of view after a token is placed on P2, T2 will not fire until t_E has elapsed, which is the same behavior of the HCN. To provide a very simple worst-case execution time analysis, we use the time requirement information associated to a component to set the lower limit of the transition firing interval and a 10% increment for the upper limit.

5. VERIFICATION WITH TINA

Listing 1, presents line 17 of the net file for the Sugar Cane system (.net PN format compatible with TINA), as a result of extracting the TPN of the system and generating its textual representation.

Listing 1: Sugar Cane TPN in net format

```
1  tr ActionEmtpy_T1 [5,6] ActionEmtpy_P2 -> ActionControlFlow13_P
```

Listing 2: Sugar Cane TPN reachability analysis

```
1  # net SugarCane, 26 places, 23 transitions                    #
2  # bounded, not live, not reversible                           #
3  # abstraction      count      props      psets       dead      live #
4  #      states         33         26          ?          2         2 #
5  # transitions         46         23          ?          5         0 #
```

The square brackets are used to indicate that the interval is closed,], or opened, [, and a w[indicates infinite. Transition `ActionEmtpy_T1` which corresponds to the *EmptyAlarm* component was assigned a local postcondition duration constraint of 5ms to keep the level comparison and respective alarm generation below 10ms (requirement SF:0006). The interval is then set to [5,6], since the interval only accepts integers.

Listing 2 presents the result of a reachability analysis performed with TINA. The first thing to notice is that the net has 2 dead states, meaning that there two deadlock states and of unfireable transitions. Taking a look at the rest of the analysis results we get a list of dead transitions: FlowActuators_T1 ControlFlow_T1 ControlDisplay_T2 ControlDisplay_T1 ActionJoinNode1_T1. With this information and the reachability graph we trace the deadlock to the join node of the action activity (figure 6). The dead lock exists because two control flows entering the join come from a control path with a decision node, i.e., only one of the two paths will carry tokens. This is fixed by adding a merge node before the join node as presented in figure 7. Further, the TPN can be used as described in [19] to perform a reachability analysis that can detect PN states that are unreachable due to time constraints (the details on how to perform this analysis are out of the scope of this paper).

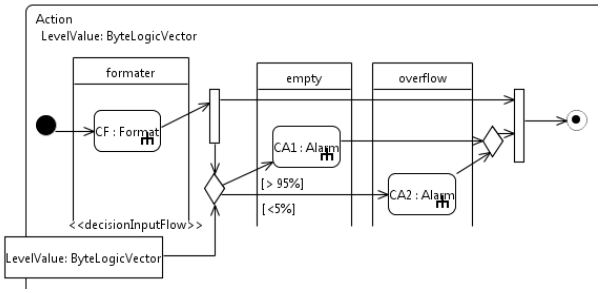

Figure 7: Fixed action component activity diagram.

6. RELATED WORK

Model-based engineering to build embedded systems is a field in expansion. Many of these works are motivated by the possibility of raising the level of abstraction of the architecture and design of the system and to perform, on these higher-level models, cost effective tasks of verification and validation.

According to the embedded design scenario presented by Gajski [7], there are four abstraction domains of an embedded system (system, processor, logic, circuit) and, at each one, a particular language can be used to describe and analyze the system. Here we do not intent to make an exhaustive comparison with the related work but to illustrate, using the Gajski framework, where is placed our work and what is its main contribution.

At the system level which is the highest, several works have been proposed for such a language and many of them are based on UML. [8] presents a comparison of some related UML approaches that include time constrains issues; some of them have analysis capabilities and tool support. Analysis can consist of checking a set of design constraints or to allow non-functional requirements to be included as part of the system design [3]. On the other hand, at the processor level we can mention at least two different approaches that focus on validation at the processor level. In the first one, named Complex [12], the MARTE profile provides capabilities to specify system functionalities and automatic virtual prototype generation at the processor level, in SystemC, is done with MDT too. Solution space exploration is done by comparing VP simulation results with system requirements. After the best solution is found, an RTL Model, i. e., at the logic level, is generated (in VHDL). In the second one, [9], UML is used to build the system level model using the COMET concurrent object structuring criteria and a VP based on Colored Petri Nets (CPN) is constructed systematically. The CPN can then be simulated and information from token flow through the CPN used to validate the system against requirements.

Our approach allows designers to specify, at the SysML level, duration constraints of the functionalities of the system. However we do not provide any type of analysis at this level. To provide system verification, the high level specification is transformed automatically, using MDT, to the processor level, similarly to Complex. However, we differentiate from the two solutions presented above in that our processor level model can be used for verification. In our case the formalism used at the processor level is HiLeS-T. Since HiLeS-T is based on PN, our approach is similar to [9]. Although the VP in Complex is constructed in SystemC it is important to note that it is built around formal models of computation (MoC) which provide improved validation capabilities.

Additionally, if the complete MTC presented in [14] is used, we can also generate a VP that can be simulated and used for validation. This prototype, either generated in VHDL-

AMS or Verilog-AMS, can be simulated to perform validation. As with Complex, this VP is based on a formal MoC (TPN). In summary, we specify at the system level, we verify some properties at the processor level and we validate at the logic level. An important aspect of our approach is that all synthesis process (i.e., model transformations to lower abstraction levels) are automated.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach to early verification of embedded systems design. We have a model based design process that has several steps supported by a model transformation chain that takes models from high level abstractions, in SysML, to models in lower levels of abstractions in hardware design languages. As an intermediate step, we use a formalism based on Timed Petri Nets called HiLeS-T. We take advantage of this step to extract the behavior information from the HiLes model into a Timed Petri Net model that can be used as an input for model checking verification. In concrete, we use Tina tools to perform reachability analysis. As we have shown in our example, by doing this step we can detect and correct early problems in the design of the systems, avoiding the otherwise not cost effective process of design, prototyping and test. By automating the process we reduce the errors associated to manually synthesizing the models. Further, the Timed Petri Net model can be used in Petri Net tools that provide advanced time analysis.

Currently we are working on other possibilities to perform early verification and validation mixing model checking with simulation techniques. In the short term, we are trying to use models to track and then to visualize in the highest level of abstractions (SysML) the problems detected in lower levels, for example in the petri net analysis or in the hardware simulation to facilitate the corrections.

8. REFERENCES

[1] B. Berthomieu, P. O. Ribet, and F. Vernadat. The tool tina - construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[2] P. Bouyer, S. Haddad, and P. Reynier. Timed petri nets and timed automata: On the discriminating power of zeno sequences. *Automata, Languages and Programming*, pages 420–431, 2006.

[3] E. P. De Freitas, M. A. Wehrmeister, E. T. Silva, Jr., F. C. Carvalho, C. E. Pereira, and F. R. Wagner. Deraf: a high-level aspects framework for distributed embedded real-time systems design. In *Proceedings of the 10th international conference on Early aspects: current challenges and future directions*, pages 55–74, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] M. dos Santos Soares and J. L. M. Vrancken. Model-driven user requirements specification using sysml. *JSW*, 3(6):57–68, 2008.

[5] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366 –390, Mar. 1997.

[6] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[7] D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design, Modeling, Synthesis and Verification*. Springer, 2009.

[8] A. Gherbi and F. Khendek. Uml profiles for real-time systems and their applications, in. *Journal of Object Technology*, 5:149–169, 2006.

[9] H. Gomaa and J. Fant. Modeling and prototyping of real-time embedded software architectural designs with colored petri nets. In *Fourth International Workshop on Model Based Architecting and Construction of Embedded Systems*, pages 85–98, 2011.

[10] C. Gómez. Modeling complex systems using sysml. Master's thesis, Universidad de los Andes, Bogotá, Colombia, 2009.

[11] C. Gómez, J. Jiménez, J. Pascal, and P. Esteban. HiLeS designer: a modeling tool for embedded systems design validation. In *5th InternationalConference on Production Research. Technologies in Logistics and Manufacturingfor Small and Medium Enterprises*, 2010.

[12] K. Gruttner, K. Hylla, S. Rosinger, and W. Nebel. Towards an esl framework for timing and power aware rapid prototyping of hw/sw systems. In *Specification Design Languages, 2010. IC 2010. Forum on*, pages 1 –6, sept. 2010.

[13] K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 37–56, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[14] H. Hoyos, R. Casallas, F. Jiménez, and D. Correal. HiLeS2: model driven embedded system virtual prototype generation. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 75–82, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[15] J. Martin. Overview of the EIA 632 standard: processes for engineering a system. In *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, volume 1, pages B32 –1–9 vol.1, oct-7 nov 1998.

[16] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *43rd annual Design Automation Conference*, pages 915–918. SIGDA and ACM, ACM, 2006.

[17] W. Tsai, F. Zhu, L. Yu, R. Paul, and C. Fan. Verification patterns for rapid embedded system verification. In *Proceedings of International Conference on Embedded Systems and Applications (ESA)*, pages 310–316, 2003.

[18] Y. Vanderperren and W. Dehaene. A model-driven development process for low power soc using uml. In *UML for SOC Design*, pages 223–252. Springer US, 2005.

[19] J. Wang, Y. Deng, and G. Xu. Reachability analysis of real-time systems using time petri nets. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 30(5):725–736, 2000.

Functional Validation of AADL Models via Model Transformation to SystemC with ATL

Pierre Bomel, Dominique Blouin, Mickael Lanoe, Eric Senn
Lab-STICC
Université de Bretagne Sud
Lorient, France
+33 (0)2 97 87 45 26
{pierre.bomel, dominique.blouin, mickael.lanoe, eric.senn}@univ-ubs.fr

ABSTRACT

In this paper, we put into action an ATL model transformation in order to automatically generate SystemC models from AADL models. The AADL models represent electronic systems to be embedded into FPGAs. Our contribution allows for an early analytical estimation of energetic needs and a rapid SystemC simulation before implementation. The transformation has been tested to simulate an existing video image processing system embedded into a Xilinx Virtex5 FPGA.

Categories and Subject Descriptors
B.7.2 [Integrated Circuits, Design Aids]: Simulation

General Terms
Design, Languages

Keywords
AADL, MDE, Program Synthesis, ATL, SystemC, Simulation, FPGA, Functional Validation.

1. INTRODUCTION

Energy. To be able to use the huge quantity of hardware resources available inside today's FPGAs, new electronic system level (ESL) design methodologies and tools are necessary. Particularly, the ever increasing density of transistors, the complexity (number of gates) of assembled hardware functions and the apparition of new 3D ICs have the consequence that energetic needs are rising, and will drastically continue to do so. This is what the IRTS revealed when it added its "Energy" chapter in its annual report [1]. Therefore, the energy consumption can prevent systems to run for long because of heat dissipation problems or fast battery discharge.

HRMPSoC. Embedded systems are becoming more and more complex. They contain computing processors (microprocessors or IPcores), memory hierarchies (caches, scratchpads, local and external memories ...), communication links (point to point, bus, NoC) and rapid IO devices (Ethernet 1Gbit, real time video, network of sensors …).

These systems can be dynamically and totally or partially reconfigurable on the fly. They are heterogeneous (a mix of hardware and software functions) and may have "time variable architectures" depending on the ability of the application to react to environment changes. These systems are called HRMPSoC (Heterogeneous and Reconfigurable Multiprocessors Systems on a Chip), have a substantial processing power, are self-adaptative and are more and more numerous in a mobile and distributed environment (so called "ubiquitous").

These systems have three important qualities: huge number of transistors, heterogeneity of implemented functions and time variable architectures. Their co-simulation (co because of heterogeneity and time variability) at high abstraction levels is required and promoted because it is necessary to validate as quickly and as soon as possible the functional correctness of several candidate architectures. These architectures are built from a set of reused or synthesized on demand components. In such context, Trabelsi et al. [2] illustrate the fact that functional validation and early estimation of energetic needs by simulation are key factors in the choice of the best architecture. Moreover, it is methodologically efficient to tie both concerns inside a common specification environment to write once and then share several times the same system models.

It is proposed to federate analytical energy estimations with functional validation of electronic systems into an up-to-date and unique modeling environment based on the Eclipse IDE (Integrated Development Environment) and the SAE (Society of Automotive Engineers) Architecture Analysis and Design Language (AADL) [3]. AADL is an emerging standard architecture description language for real-time, fault-tolerant, scalable and embedded multiprocessor systems. It is component-centric and allows specifying both software and hardware parts of systems. A SystemC model is built by automatically assembling components previously grouped in a library in compliance with the architecture specified with AADL. Thus, having a unique AADL model of an FPGA based system helps designers to check two important constraints: 1) that the energetic needs do not exceed a given value, and 2) that the assembled system is functional.

This paper presents our work related to automatic generation of SystemC models from AADL models. Our automatic generation takes advantage of model transformation, which is expressed with the ATL language [4]. In section 2 we present the state of the art in the domain of automatic generation of models from AADL specifications. In section 3 we present our contributions: a methodology, a semantic mapping between meta-models elements of AADL and SystemC languages and finally a set of ATL transformations. We validate our contributions in section 4 with a video processing system model. We conclude in section 5.

2. RELATED WORK

AADL enables the development and predictable integration of highly evolvable systems as well as analysis of existing systems. It supports early and repeated analyses of system architectures with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined

semantics. In this section we inventory related work about analysis and/or generation of executable models, with or without the use of MDE techniques, from such AADL models. Most of this work concerns the verification of functional and non-functional system properties or the validation of systems by co-simulation in order to extract temporal estimations dynamically without the need for ISS (Instruction Set Simulators) and RTL level models like in complex and long simulations.

Ocarina [5]: Ocarina is a software tool which allows putting into action an evolutionary prototyping methodology based on AADL. Worst case execution time and dead-lock freedom are some of the non functional properties it can check. It also generates ADA or C executables on top of the high integrity POLYORB-HI middleware, in turn targeting ERC32 and LEON2 processors.

Cheddar [6]: Cheddar is an open source tool developed in ADA. With the help of simulations, it computes various performances criteria (schedulability analysis, time constraints, resources allocation, etc.). It accepts as input AADL models thanks to its embedded Ocarina API. Given the difficulties to apply schedulability theory, the authors have recently decided to exploit an MDE methodology to automatically generate, with the help of the Platypus tool, some decision support tools that will determine the relevant feasibility tests for a given architecture to evaluate. Platypus is a meta-model environment relying on the STEP standard (ISO 10303, EXPRESS language).

ACSR [7] and VERSA: The University of Pennsylvania, in collaboration with the Freemont Company, has developed a code generator that translates an AADL model into an ACSR model (Algebra of real-time process). This ACSR model can be analyzed with a tool in order to conduct schedulability analysis.

OSATE [8]: OSATE is a set of Eclipse plugins for the modeling of embedded electronic systems in AADL. It is based on EMF and contains a complete AADL meta-model. OSATE, as an extension of Eclipse, is itself an environment for integrating other tools that operate on AADL. Version 1.5, used for our work incorporates many analysis tools, but no real tools for code generation for executable models.

TASTE [9]: The TASTE toolset is the result of work of the ASSERT (IST 004033, 2004-2007) European project. It was developed by ESA (European Space Agency) with a set of partners in the aerospace field. It aims to define a development process of distributed real-time systems and is based on a tool chain which includes Cheddar and Ocarina. TASTE can build a system from heterogeneous software (MathLab, Ada, C, C++ ...). These codes are either generated automatically by using external tools or manually written. The overall system consistency is ensured by the use of two modeling languages: system modeling with AADL, and messages/data modeling between heterogeneous modules with ASN.1. Code generators are used during the modeling phases to produce software for a given target. TASTE does not generate a mixed executable model for co-simulating hardware-software. Neither does it currently include hardware features, although it seems to be part of future extensions.

Gaspard2 [10]. Gaspard2 is a modeling environment for real-time systems dedicated to intensive and regular data processing. These processes can be represented using a formalism derived from ARRAY-OL whose semantics has been adopted in the UML MARTE profile. It can generate a SystemC 2.0 TML-level simulation model. This model is based on the notion of virtual processor and allows representation of both software and hardware features. Finally, it incorporates the estimated consumption in the SystemC simulation model. However it does not accept AADL models as input and does not offer an analytical model to estimate the power consumption.

AADS, SCOPE [11]. AADS is a tool written in Java for the hardware/software co-simulation environment named SCOPE. It converts an AADL model into a SCOPE model. The SCOPE model is compatible with the Ravenscar computation model. SCOPE is a co-simulation environment written in SystemC, which provides time information on the various system tasks. To do this, no instruction sets simulator is used but time is estimated by executing an annotated native code. It specifically targets MicroC and POSIX OS operating systems and the LEON2 processor.

Apart from AADS, none of the work cited above does target both SystemC code generation and AADL modeling. One of the two languages is always missing. Finally, AADS does not use the MDE methodology to convert an AADL model into a SystemC model. Our contribution is to implement a model transformation in a standardized modeling environment (OSATE) targeting another standardized and highly flexible simulation environment (SystemC, IEEE 1666-2005).

3. CONTRIBUTIONS

In this section we present our design methodology, the set of AADL/SystemC semantic mappings and the ATL model transformations supporting the automated generation process.

3.1 Methodology

The methodology that we propose belongs to the category of "fast and evolutionary prototyping" [12]. It is based on a combination of modeling techniques, code generation and evaluation. It is shown in Figure 1 and is divided into six phases:

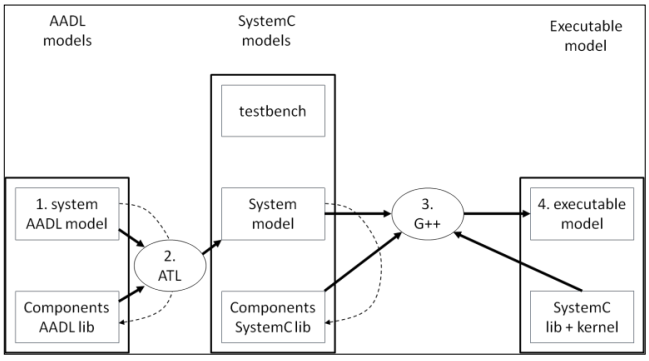

Figure 1. Model/Generate/Simulate methodology flow.

1. Use a library of components to model a system with AADL/OSATE. This library is enriched by
   - IP designers that provide AADL and SystemC models,
   - and sub-systems previously modeled, generated and validated.

2. Automatically generate the complete system model in SystemC by means of a chain of ATL transformations. A simplified meta-model for SystemC has been developed including only the necessary concepts needed for C++ code generation from SystemC models.

3. Integrate the generated SystemC model in the system architect's test program. To do this, simply compile the code generated from the SystemC models of the

assembled components and the test program, then link all with the SystemC simulation kernel.

4. Simulate the complete system with the resulting executable. The architect judges the validity of the system in light of the results based on provided inputs and expected outputs.

5. If the system is considered functional, the designer can estimate the energy consumption. But, he may as well start with the energy estimation and then check the functionality second. Energy estimation is performed using analysis models whose input often depend on both software and hardware parameters. Besides functional validation, SystemC simulation can also be used to obtain estimates for some of these input parameters.

6. When the system is functional and "energy correct", we can then move on to the detailed design phase or repeat this method to evaluate a different architecture, or the same architecture with another components library. The amount of effort needed for the detailed design phase depends on the available component libraries. If RTL components already exist, they can be reused. Otherwise, they must be developed, which may require significant efforts.

The two dashed arrows in Figure 1 indicate that the obtained AADL and SystemC models can be respectively added to the AADL models libraries and SystemC components library. This methodology allows the building of libraries of increasingly complex components.

The components are initially designed to represent a computable artifact of the behavior of functions. They do not necessarily represent their final implementation. As such, they can represent both hardware or software functions. Anyway, there is nothing that prevents the existence of several SystemC models of the same function. Therefore, they could represent the same function with different implementation types or different abstraction levels and, as long as their interface with the system remains the same, they can coexist in the libraries.

3.2 AADL / SystemC Mapping

AADL permits the modeling of an electronic system in terms of software and hardware components 1) which communicate with each other and 2) with the placement of interconnected software components over the hardware execution platform. The hardware is itself made out of a set of connected hardware components

In the scope of our methodology, the objective is the rapid functional validation of a components assembly, each component having a functional representation in SystemC. The AADL subset we have chosen for this methodology allows the description of data types, interface components, system architectures, shared data, and communications between components and the external interface of the complete system. The link between AADL and SystemC entities is defined thanks to annotations added in the AADL model. Finally, the model transformation must consider the incompatibilities between the rules for naming identifiers. Unambiguous AADL to SystemC conversion rules are needed.

Data types. All types of data processed by components have matched AADL and SystemC models. Let *CppX* be the name of a C or C++ data type, and *AadlY* the name of the corresponding AADL data type. Then the AADL data type *AadlY* has the form shown in Figure 2:

```
data AadlY
  properties  Type_Source_Name => "CppX";
end AadlY;
```

Figure 2. AADL model of a data type.

The AADL model is reduced to the creation of an AADL component of type *data* with the name *AadlY*. The value of the property *Type_Source_Name* is the annotation that indicates the semantic mapping between *CppX* and *AadlY*.

Components. Our AADL components are black boxes for which only the interface is known. They are represented by AADL *threads*. Their interface consists of communication *ports* and *accesses* to shared data.

As shown in Figure 3, the AADL model contains a description of a *thread* and its *implementation*. Inside its *features* section, the *thread* contains a list of ports of type *event data port* when some typed data transit and of type *event port* when it comes to digital only signals. It also contains a list of shared variables that it must have access to. This is expressed via a *requires data access* clause. The mapping with the SystemC module *CppThread*, which represents the true functionality of the *thread,* is declared with an annotation: we use the value of the property *Source_Text* in the *implementation* of the *thread*. Note here the implicit identity between the AADL ports and SystemC ports of both models. Finally, the notion of shared data is also implicitly synonymous to a C++ global variable that is shared by the codes of the SC_METHOD or SC_THREAD SystemC processes declared in the SC_MODULE.

```
thread AadlThread
  features
    id : in  event data port AadlY;
    od : out event data port AadlY;
    i  : in  event port;
    o  : out event port;
    d    : requires data access AadlY;
end AadlThread;

thread implementation AadlThread.impl
  properties  Source_Text => "CppThread";
end AadlThread.impl;
```

Figure 3. AADL model of a functional component.

Architecture, Shared Data and Communications. To represent the functional architecture of the system, we use an AADL component of type *process* and its associated *implementation*. We declare in the *subcomponents* clause of the *implementation* as many threads subcomponents as we need as well as all the shared data subcomponents that *threads* need to read/write from. Finally we connect the *ports*. Figure 4 illustrates the architecture of such a *process* inside which N *threads* of type *AadlThread* are chained together and the ends of the chain are connected to the *ports* of the *process*. It also creates the shared data *d*, of type *AadlY*, and indicates that all *threads* have access to it.

System Interface. The complete top level system is modeled using an AADL *system* component type. It has the same type of interface than the assembled components. The *implementation* of the system declares an instance of the *process* modeled earlier and connects its ports to those of the top level system (Figure 5).

The identity of the interface of AADL components of type *process* and *system* allows for repeatedly enriching the libraries from the AADL modeling process. During the generation of SystemC modules, the same top level system name is created and becomes a reusable and valid SC_MODULE. This name will be available for future annotations via the *Source_Text* property.

Thus, *AadlSyst* is a module that can be added to the SystemC components library and can be reused for future AADL models.

```
process SystemArch
  features
    id : in  event data port AadlY;
    od : out event data port AadlY;
    i  : in  event port;
    o  : out event port;
end SystemArch;

process implementation SystemArch.impl
  subcomponents
    t1  : thread AadlThread.impl;
    …
    tN  : thread AadlThread.impl;
    d   : data AadlY;
  connections
    c1a : event data port id   -> t1.id;
    c1b : event port      i    -> t1.i;
    d1  : data access     d    -> t1.d;
    …
    cNa : event data port t(N-1).od -> tN.id;
    cNb : event port      t(N-1).o  -> tN.i;
    dN  : data access     d         -> tN.d;
    cNc : event data port tN.od     -> od ;
    cNd : event port      tN.o      -> o ;
end SystemArch.impl;
```

Figure 4. AADL model of the architecture of the system.

Refinement and Implementation. The AADL concepts of refinement (*refines*) and *implementation* are both naturally represented in the generated SystemC models by the C++ mechanism of inheritance.

```
system AadlSyst
  features
    id : in  event data port AadlY;
    od : out event data port AadlY;
    i  : in  event port;
    o  : out event port;
end AadlSyst;

system implementation AadlSyst.impl
  subcomponents
    arch : process SystemArch.impl;
  connections
    c1 : event data port id      -> arch.id;
    c2 : event data port arch.od -> od;
    c3 : event data port i       -> arch.i;
    c4 : event data port arch.o  -> o;
end AadlSyst.impl;
```

Figure 5. AADL model of the top level interface.

Transformation Rules for Identifiers. SystemC is a language sensitive to uppercase and lowercase while AADL is not. So, `ABCD` and `abcd` are identical in AADL, but not in C or C++. We need to agree on rules for processing AADL identifiers into new identifiers that:

- are legal in C++,
- are not identical to C, C++ or SystemC reserved keywords and macros,
- and are never duplicated.

For this, we followed the AADL SAE's recommendations about the C language [13] and have extended them to the case of SystemC and C++. They are listed here.

- The AADL namespace exists. It contains the names of all executable objects that are equivalent to AADL concepts. These names are located in a SystemC runtime library that contains all types and all classes required for the generation of C++ and SystemC models.
- To every AADL package corresponds a C++ namespace. As an example, Figure 6 shows an AADL package named *AadlPack* in which all components, data types and systems mentioned in this article are defined.
- All AADL identifiers are converted to lowercase and a mechanism for automatic prefixing with "PREFIX_" avoids duplications or collisions with keywords of C, C++ or SystemC. In addition, all characters "." are replaced by "_DOT_", and all sequences "::" are replaced by "_PATH_". Figure 7 shows all possible translation cases.

```
package AadlPack
  data AadlY …
  end AadlY;
  thread AadlThread …
  end AadlThread;
  …
  system AadlSyst …
  end AadlSyst;
  system implementation AadlSyst.impl …
  end AadlSyst.impl;
end AadlPack;
```

Figure 6. AADL package.

```
Idf             -> idf
IDF             -> idf
break           -> PREFIX_break
a.b             -> a_DOT_b
c_DOT_d         -> c_DOT_d
c.d             -> PREFIX_c_DOT_d
a::b            -> a_PATH_b
c_PATH_d        -> c_PATH_d
c::d            -> PREFIX_c_PATH_d
```

Figure 7. Identifier conversion examples.

3.3 ATL Model Transformations

A chain of five model transformations in ATL has been developed to generate the SystemC model. In any case, at least two transformations were needed for first transforming the AADL model into a SystemC model, and then the SystemC model into C++ code. Breaking the transformation into smaller pieces allowed reducing the complexity of the global transformation.

These transformations are based on a source AADL meta-model and a target meta-model named scMM, which is the C++ subset that represents our minimum needs to generate SystemC models. It is smaller and easier to manage than a full set of C++ and SystemC meta-models syntactically complete. Because we do not target all the C++ and SystemC specificities like compilers do, we do not require a complete meta-model. Moreover the genericity of scMM allows us to retarget to any other object-oriented language. Figure 8 shows the scMM meta-model. The C++ concepts are namespaces (*Namespace*), classes (*ClassList*, *Class*, *ClassSection* and *ClassMember*), identifiers and builders of connections (*ConnectionId*, *ConstructorConnectionInit*, and *Binding*) and finally the identification of the system model (TopLevel).

The five transformations are (Figure 9):

- *a2s.atl* is the essential exogenous transformation that converts our AADL subset into its scMM equivalent.

- *updateRefs.atl* and *updateRef2.atl* are two endogenous scMM model transformations updating internal references that could not be computed in the initial processing by a2s.atl.

- *orderClasses.atl* is the endogenous transformation whose role is to sort all classes and types in an order consistent with a compilation process.

- *sc2txt.atl* is the transformation that converts the scMM model, with all its internal references properly updated and rearranged into a compilable ASCII text. It supports the syntax of the C++ object-oriented target language.

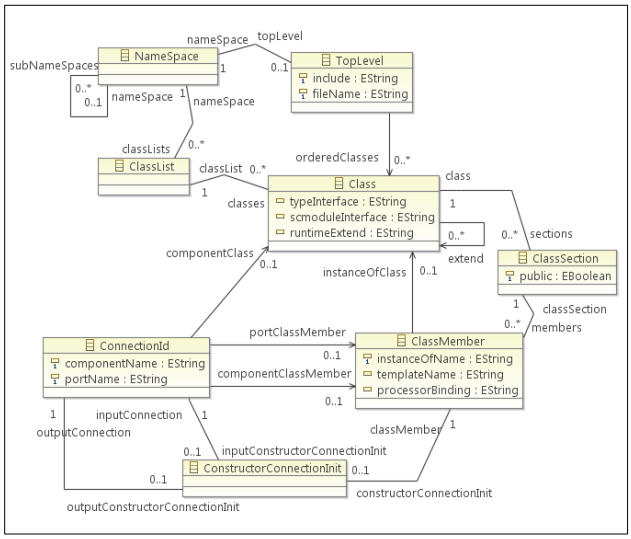

Figure 8. scMM meta-model.

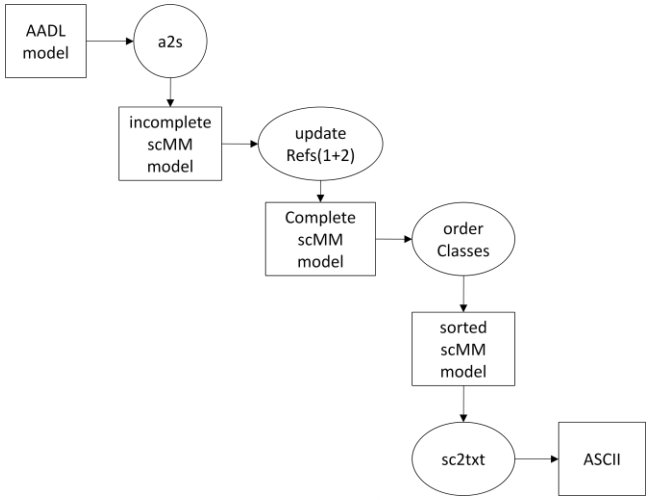

Figure 9. ATL model transformation chain.

4. RESULTS

The presented results have been tested in the following technical context: Eclipse 3.6, ATL 3.1.1, AADL/OSATE 1.5, SystemC 2.2.0 and Eclipse C Development Tools (CDT) 7.0.2. Our transformation chain has been integrated in the Eclipse IDE as a plugin whose code was partially generated by the ATL development toolkit. Users can select the AADL files to be transformed, and a directory of a predefined CDT project into which the generated C++ files will be put, properly configured for SystemC for quick simulation of the system.

We have modeled an existing image processing system with AADL that can process a 25 frames/s VGA video image stream. It is embedded into a Xilinx Virtex5 FPGA. Image capture and display are performed by hardware blocks respectively interfaced with a camcorder and an LCD screen. The image real-time processing is performed by a program executed by a synthesizable MicroBlaze processor. This system can be easily customized and serves many research and project activities. It has been developed thanks to the MOPCOM project.

For didactic purposes (black and white paper print) we have chosen to reverse the three color components (RGB) of the received images. We have transmitted the image of Lena as a very well known test input so that readers feel familiar with the presented results.

Figure 10 and Figure 11 show the graphical and AADL architecture of the system. It consists of four components whose names are meaningful: capture, processing, display and global synchronization. The synchronization block performs the permutation of the images accesses indices and schedules the image processing at a given frame rate. Capture and display blocks operate at the pixel clock. A shared memory stores a buffer of three images inside which the blocks can make reads and writes through a shared bus. In the AADL model, one can see the instantiation of the four components *Synchro0*, *Capture0*, *Display0* and *Processing0*, the *imageArray* image buffer, and the connections needed to connect the *ports* and provide access to *imageArray* to all *threads*.

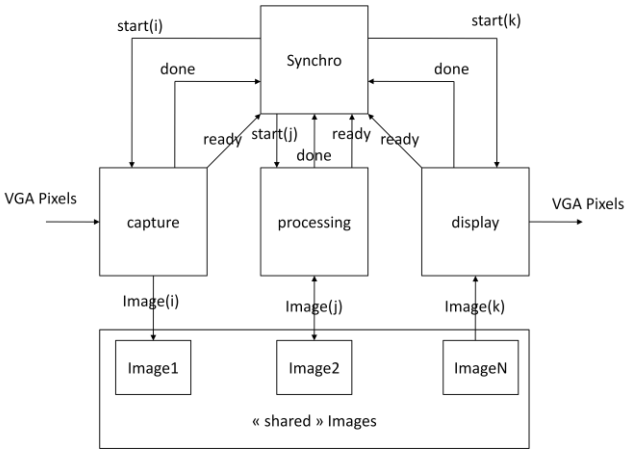

Figure 10. Video processing, system architecture.

The simulation of this architecture proves that the system is functional. The resulting images are depicted in Figure 12. While the real system is a real-time one running at a rate of 25 frames / sec, the SystemC model is simulated at a rate of only one image every four to five seconds. So we have a ratio of about 100 between the simulation speed and the real time processing rate expressed in images per seconds.

5. CONCLUSION AND PERSPECTIVES

In this article we presented our work about the transformation of AADL models into SystemC for electronic systems embedded into FPGAs. Our contributions, which have been validated by the modeling of a real time image processing system, the code generation and the SystemC simulation (consistent with the expected behavior), show that it is possible and efficient to combine in the same Eclipse meta-modeling environment the

analytical estimation of power consumption and the functional validation by simulation. By reusing the same models, the two methodologies reduce the modeling efforts imposed to the system architect. Finally, this rapid generation and simulation design process allows considering a broader exploration of the architectural design space.

```
process SoftwareArchitecture
  features
    pixel_in: in event data port VGAPixelType;
    pixel_out: out event data port VGAPixelType;
end SoftwareArchitecture;

process implementation SoftwareArchitecture.impl
  subcomponents
    Synchro0   : thread Synchro.impl;
    Capture0   : thread Capture.impl;
    Processing0: thread Processing.impl;
    Display0   : thread Display.impl;
    imageArray : data   ImageArrayType;
  connections
    pixels_in       : event data port pixel_in       -> Capture0.pixel;
    pixels_out      : event data port Display0.pixel -> pixel_out;

    capture_ready   : event port Capture0.ready    -> Synchro0.capture_ready;
    processing_ready: event port Processing0.ready -> Synchro0.processing_ready;
    display_ready   : event port Display0.ready    -> Synchro0.display_ready;

    capture_start   : event data port Synchro0.capture_start    -> Capture0.start;
    processing_start: event data port Synchro0.processing_start -> Processing0.start;
    display_start   : event data port Synchro0.display_start    -> Display0.start;

    capture_done    : event port Capture0.done    -> Synchro0.capture_done;
    processing_done : event port Processing0.done -> Synchro0.processing_done;
    display_done    : event port Display0.done    -> Synchro0.display_done;

    -- images accesses
    Capture0_image   : data access imageArray -> Capture0.images;
    Processing0_image: data access imageArray -> Processing0.images;
    Display0_image   : data access imageArray -> Display0.images;
end SoftwareArchitecture.impl;
```

Figure 11. AADL model of the system internal architecture.

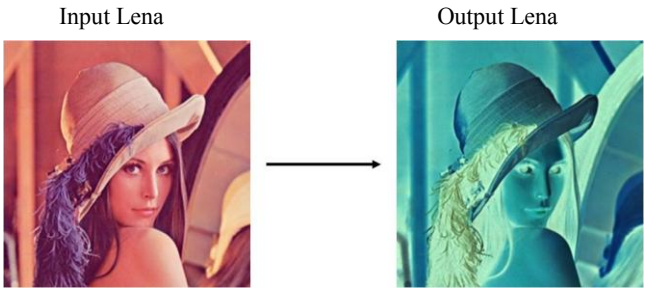

Figure 12. Simulated processed images.

During this work, we have identified that the use of incomplete AADL specifications (keyword *refines*) enables a generic modeling and a late binding mechanism during the modeling process. This mechanism seems very close to the C++ *template* concept. We intend to study it and integrate it in the ATL transformations. With this modeling feature, it will be possible to model generic architectures and refine them only when needed. Hence, functional components AND generic architectural components will be both available in our AADL and SystemC libraries.

6. ACKNOWLEDGMENTS

This work is part of the Open-PEOPLE project (26/12/2008-25/12/2011) and is currently funded by the French Research Agency (ANR). It is labeled by the “Images et Réseaux” ("Media and Networks”) Brittany pole. The MOPCOM project has been supported by the ANR (contract 2006 TLOG 022 01), the “Images et Réseaux” pole and the Bretagne and Pays de la Loire regions.

7. REFERENCES

[1] "International Technology Roadmap for Semiconductors, 2010 update", www.itrs.net

[2] C. Trabelsi, R. B. Atitallah, S. Meftali, J.-L. Dekeyser and A. Jemai, "Model-Driven Approach for Hybrid Power Estimation in Embedded Systems Design", EURASIP Journal on Embedded Systems, vol. 2011, id. 569031, Hindawi Publishing, 2011.

[3] SAE. "Architecture Analysis & Design Language" (AADL). AS5506A.

[4] F. Jouault and I Kurtev. "Transforming models with ATL". Satellite Events at the MoDELS 2005 Conference, 2005, pp. 128-138. http://wiki.eclipse.org/M2M/ATL

[5] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. "From the prototype to the final embedded system using the Ocarina AADL tool suite". ACM Transactions on Embedded Computing Systems, TECS 2008, vol. 7, n° 4, article 42, july 2008.

[6] M. Kerboeuf, A. Plantec, F. Singhoff, A. Schach and P. Dissaux. "Comparison of six ways to extend the scope of Cheddar to AADL v2 with Osate". 5th international workshop on AADL and UML. Oxford, UK, March 2010, pp. 367-372

[7] O. Sokolsky, I. Lee and D. Clark. "Schedulability Analysis of AADL models". Procs of the 20th Intl. Parallel and Distributed Processing Symposium, IPDPS 2006, vol. 2006, april 2006, article 1639421.

[8] OSATE. Carnegie Mellon Software Engineering Institute (SEI), "Open Source AADL Tool Environment" (OSATE), http://www.aadl.info/aadl/currentsite/tool/osate-down.html

[9] TASTE. M. Perrotin, E. Conquet, P. Dissaux, T. Tsiodras and J. Hugues. "The TASTE Toolset : turning human designed heterogeneous systems into computer built homogeneous software". Procs. of the Intl. Conf. Embedded Real Time Software Systems (ERTSS), may 2010, Toulouse, France.

[10] R. Ben Atitallah R., E. Piel, S. Niar, P. Marquet and J.-L. Dekeyser J-L. "A Fast MPSoC Virtual Prototyping for Intensive Signal Processing Applications". Microprocessors and Microsystems Embedded Hardware Design Journal (MICPRO) 2011.

[11] Roberto Varona, Eugenio Villar and A-I. Rodríguez. "Ravenscar Computational Model compliant AADL Simulation on Leon2". Procs. of the International Symposium on Information System and Software Engineering, ISSE 2011, March 2011, Prague, Czech Republic.

[12] F. Kordon and Luqi. "An Introduction to Rapid System Prototyping". IEEE Transactions on Software Engineering, 70(3), pp. 817-821, 2002

[13] SAE, "Language Compliance and application program Interface". The AADL specification, 2005, annex volume 1, annex D.

Automatic SysML-based Safety Analysis

Philipp Helle
EADS Innovation Works
Nesspriel 1
21129 Hamburg, Germany
philipp.helle@eads.net

ABSTRACT

Model-based Safety Analysis (MBSA) techniques exist that ensure an increased consistency by formalising the safety analysis and allow automation of the safety calculations. With the increased acceptance of Model-based Systems Engineering (MBSE) as the new systems engineering paradigm, it seems natural to combine MBSE and MBSA.. This work provides a methodology and tool support for an integrated MBSE and MBSA on one common model based on SysML[6] which allows the systems engineers to perform an automated safety analysis to receive quick feedback on their design decisions during the system design phase.

Keywords

SysML, MBSE, Safety Analysis, MBSA

1. INTRODUCTION

Increasing system complexity results in an increase in complexity of the safety analyst's task to ensure that systems are safe. Model-based Safety Analysis methods have been developed for formalising the work and subsequent automation of the safety calculations. However, these techniques use their own models that are not identical to the design models. Keeping consistency between these models either requires manual effort or model-to-model transformations. Additonally, the system design is often separated from the safety analysis process, the connection often being an "over the wall process"[5], and the system designers receive the safety analysis results late in their work which makes necessary changes more expensive.

Given that Model-based Systems Engineering [1] is increasingly accepted and employed in industry as the new systems engineering paradigm, it seems natural to combine MBSE and MBSA. The key idea of this work is to extend the SysML[6] MBSE design model to include safety related aspects. This will allow the systems engineers to perform an automated safety analysis to get feedback on their design decisions without the need for a safety specialist.

Paper structure: Section 2 provides background information regarding system safety. Section 3 defines the rules for safety calculations used in this work and section 4 introduces the concept of model-based safety analysis and provides exampels for existing implementations. Section 5 describes our concept for an automated safety analysis based on SysML models. The process and the modelling extensions will be illustrated by a running example. Section 6 provides a conclusion to this paper.

2. SYSTEM SAFETY

System safety uses systems theory and systems engineering approaches to prevent foreseeable accidents and to minimize the result of unforeseen ones. It is a planned, disciplined, and systematic approach to identifying, analyzing, and controlling hazards throughout the life cycle of a system in order to prevent or reduce accidents [14].

2.1 Safety Terminology

Unfortunately, safety and reliability are sometimes used interchangeably. Risk is "[a] combination of the likelihood of harm and the severity of that harm" [16]. Safety is defined as "freedom from unacceptable risk" [17] and reliability is defined as "the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions" [14]. So, whereas reliability deals with all potential failures, safety only deals with the hazardous ones [13].

Generally, safety has a wider scope than failures, and failures do not necessarily compromise safety. Many accidents occur even when the individual components were operating exactly as specified or intended, that is, without failure. The opposite is also true - components may fail without a resulting accident.

2.2 Safety engineering process

Existing safety engineering processes are normally based on standards that define a common framework for the derivation of safety requirements which combines hazard assessment and risk analysis techniques. The aim of the analysis

[1]The International Council on Systems Engineering defines MBSE as "the formalised application of modelling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases"[7].

is to determine:

- The critical system functions, i.e. functions which may cause a hazard when lost or malfunctioning
- The safety requirements for these functions, i.e. the maximum allowed failure probabilities.
- The demands, if any, for additional safety functions in order to achieve acceptable levels of risk for the system.

Figure 1 depicts the steps in the development of an aircraft system and the related safety assessment procedures. It is based on the two main standards relating to safety in the civil aerospace[2] domain: ARP-4754[10] and ARP-4761[11].

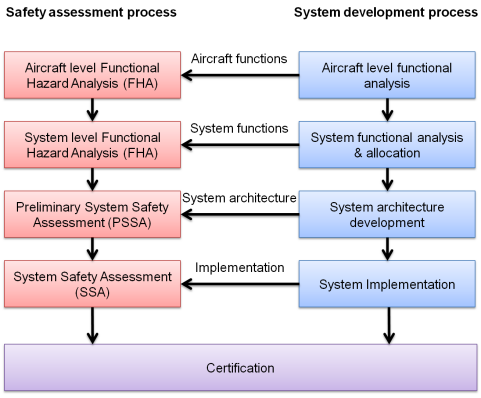

Figure 1: Safety assessment process in relation to the systems development process

2.3 Classical Safety Analysis Methods

Various causally based techniques for systems safety assessment based in known designs have evolved. Usually, these fall into two classes - methods which work from known causes to unknown effects (such as Failure Modes and Effects Analysis) [19] or those which work from known effects back to initially unknown causes (such as Fault Tree Analysis) [19].

The target usage of these techniques varies depending on the domain and the nature of the problem. For example, fault trees are commonly used in the civil aerospace domain at the Preliminary System Safety Assessment (PSSA) phase to examine whether the system can achieve the safety requirements allocated from the hazard identification [13].

While there exist tools supporting these classical safety analysis methods, most of the work is still done manually.

3. SAFETY CALCULATIONS

This section explains the rules for safety calculations. The probability of a failure event "component c failing" is denoted as *Pc*. The inverse event "component c is functional" then is *1 - Pc* denoted as *Qc*. For the derivation of the failure formulas it is assumed that all failure events are independent of each other. Let there be two components *c1* and *c2* with their respective probabilities *Pc1*, *Qc1* and *Pc2*, *Qc2*. Then, the probability of both events occurring is *Pc1 * Pc2* [15].

The probability that one of both components fails is $P(A|B) = P(A) + P(B) - P(A\backslash B)$. This can be further reduced under the assumption that the failure probabilities of components are typically of the order 10-3 or smaller. From this follows that the term $P(A\backslash B)$ is very close to zero and can be omitted without significantly loosing accuracy. Hence, the probability of *c1* or *c2* occurring is *Pc1 + Pc2* [15].

Failure expressions can be simplified with the help of the idempotence law and the absorption law. For instance, let there be three components *c1*, *c2*, *c3* and three failure events A, B, C with A = "c1 fails", B = "c2 fails" and C = "c3 fails". Assume the following failure expression (using + and * for the set operations) $A + ((A + B) \mid (A + C))$. Using the rules of boolean algebra, this expression is equivalent to the term $A + AA + AC + AB + BC$. Applying the idempotence law yields the term $A + AC + AB + BC$ which finally can be simplified to $A + BC$ with the help of the absorption law. Expressing this algebraic term in terms of component failures this means, that component *c1* fails, or the components *c2* and *c3* fail together.

3.1 Reliability Block Diagrams

Reliability Block Diagrams (RBD) allow visually representing the success logic of a system by using block structures that are connected with each other via success paths. A success path is a "set of components which, when working, connect the start node with the end node [of a system] through working components thereby guaranteeing that the system is in working state" [20]. A minimal path is a path "from which no component can be removed without disconnecting the link it creates between the start and the end node" [20]. It holds that the elements required for the system are connected in series, while elements that can fail without affecting the system to work are connected in parallel [2].

3.2 Minimal Cut Sets

With the help of analytical methods RBDs can be evaluated to calculate the system reliability [21]. But, since an RBD can contain each component more than once, the failure probabilities cannot be determined by summing up the probabilities within a channel and multiplying the channel probabilities. Instead, a so called minimal cut set of the RBD has to be found that contains each element only once. A cut set is a "set of components which, when failed, disconnect the start node from the end node and the system is in a failed state"[20]. A minimal cut set is a "cut set for which no component can be returned in working state without creating a path between the start node and the end node, thereby returning the system into a working state"[20]. Using the safety calculation rules introduced in section 3 it is possible to calculate the failure probability of the system, even if there are multiple occurrences of a component in several success paths.

3.3 Failure modes

Failure modes describe how components can fail[12]. They are commonly divided into *content failures*, which means

[2]The methodology was developed and is therefore embedded in an aeronautic context but generally is independent of a specific domain

that the components delivers data in a way that deviates from the assigned functionality, and *timing failures* which means that the component sends its data at the wrong point in time. This can either be too early, including sending data when none is required, or too late.

In the context of this work, content failures are referred to as *loss failures* because the loss of a functionality in a broader sense can also be seen as an "deviation from the assigned functionality". Timing failures, on the other hand, are referred to as *spurious failures*[3].

Depending on the failure mode, the calculation of a failure case differs. When calculating the probability of a loss failure for a function, the function's dependencies are considered. Calculating the probability of a spurious failure for a function, however, does not consider the functional dependencies of that function, but the functions that are dependent of it because the effect of an inadvertently sent signal on these functions has to be determined.

4. MODEL-BASED SAFETY ANALYSIS

4.1 Existing solutions

To automate safety activities and also to extend and complement the classical safety analysis techniques, a variety of formal safety analysis techniques exists. One of the most prominent examples is the AltaRica [1] language. AltaRica models formally specify the behaviour of systems when faults occur. These models can be assessed by means of complementary tools, e.g. to calculate overall failure rates and derive fault tree diagrams automatically. The safety model of a system in AltaRica is not identical to the design model of that system in e.g. SysML. Keeping consistency between these models either requires manual effort or model-to-model transformations. Additionally, these models are built typically after the system design and thus necessary changes that result from the safety analysis are costly to put into effect.

4.2 Integrating MBSE and MBSA

With the increased acceptance of MBSE as the new systems engineering paradigm, it seems natural to combine MBSE and MBSA. One possibility is to automatically extract minimal cut sets directly from detailed design models bypassing Fault Tree generation altogether. This approach [3] allows truly automated analysis of detailed design models thus minimising both the possibility of safety analysis errors and the cost of the analysis. The application scope of the approach is the detailed design phase of a system and it requires detailed component models to work.

Another possibility is to derive models suitable for safety analysis from the system development models. In [4] provides an example for deducing analyzable AltaRica code from UML/SysML models. This approach requires the availability of detailed AltaRica node implementations for all involved system components.

In contrast to these described existing methodologies, which represent a number of further research work that has a similar direction, e.g.[8], our approach is explicitly light-weight and targets system design in its early phases. The idea is that the system designer can run an approximated safety analysis during the design process without the necessary aid of a safety specialist to get very quick feedback on design decisions.

[3]Powell [18] uses the terms *value* and *timing errors*.

5. AUTOMATIC SAFETY ANALYSIS

For our approach, we extend the already existing meta model [9] for functional and systems architecture modeling with concepts from the safety domain as depicted by Figure 2. Textual safety requirements are formalized as failure cases with attributes that define the maximum allowed probability for a failure case to occur. The failure case in turn is defined by its relation to one or more functions which have to fail in order for the failure case to occur. Note, that the function(s) that the failure case is related to serve as a starting point for recursively propagating that failure in the functional architecture. The basic assumption here is that a function fails when one or more of the functions that it needs input from fail. Additional modifications on the relation between the failure case and the function, e.g. the definition of the failure mode, allow modifying how this propagation is done, e.g. propagation without restrictions, propagation up to a certain depth or no propagation at all.

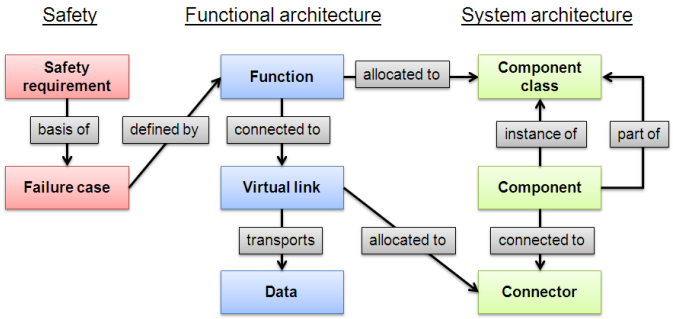

Figure 2: Safety meta model

5.1 Running example

The envisaged automated safety analyis process and the modelling notations used will be demonstrated using a running exanpmle. The Fire Detection System (FDS) is a simple system for the detection and (visual) warning in case of fire. It consists of a busbar, a fire warning lamp, and a number of fuses and fire detectors.

5.2 Modelling process

The envisaged safety analysis process is depicted by 3. It illustrates how the traditional safety and engineering process steps and their connections (see Figure 1) are extended to include the automatic system safety analysis process step. Note, that this belongs to the system development process, i.e. it can be performed by the systems engineers in their daily work without the involvement of a safety analyst or a dedicated safety analysis tool. This enables instant feedback on design decisions.

5.3 Modelling notation

This subsection describes the modelling formalisms that are used to model the system and the safety related extensions

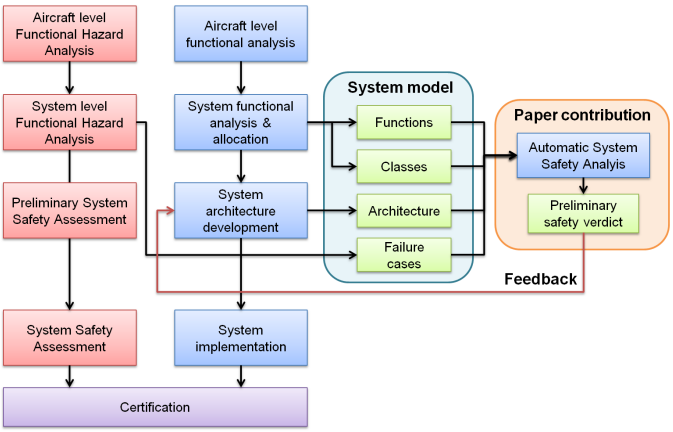

Figure 3: MBSA Process

such as failure cases. The functional architecture is realized using SysML Activity Diagrams. Functions are represented by Actions and functional dependencies as Data Flows and/or Control Flows.

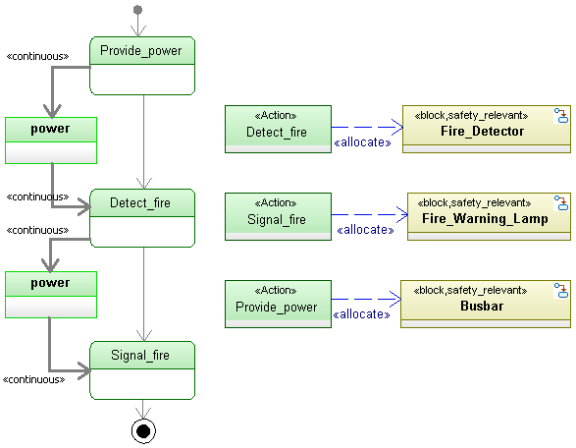

Figure 4: FDS functions

System components, modeled as blocks, are extended by an additional stereotype «*safety_relevant*», which allows the annotation of the blocks with characteristics that are needed in the context of safety analysis such as the failure rates for the different failure modes.

Actions, representing functions, are allocated to the classes via an «*allocate*»dependency. Figure 4 shows on the left side the three identified functions of the FDS, their sequence and the data flows between them and the allocation of the functions to system components on the right side.

Failure cases are defined by use cases marked by the stereotype «*Failure_Case*». The formal semantics of a failure case is defined by dependency associations that relate functions to failure cases. These dependency associations must be assigned a stereotype corresponding to a failure type: for loss failures this stereotype is «*loss*», for spurious failures «*spurious*». The dependencies can be shown graphically on SysML Use Case Diagrams as demonstrated by Figures 5 and 6.

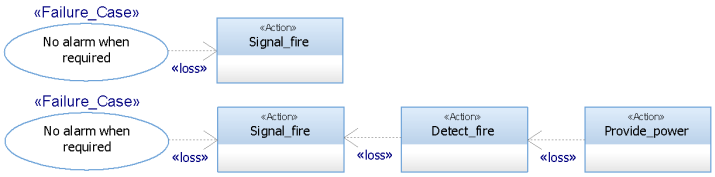

Figure 5: FDS failure case 1 - before and after expansion

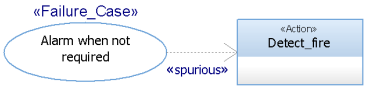

Figure 6: FDS failure case 2

Additionally, each failure case has a defined maximum allowed failure rate, which defines what rate of failure is acceptable in a system for the given failure case. For the FDS the maximum rates are 2x10-5 for *No alarm when required* and 1x10-5 for *Alarm when not required* for one hour operating time.

The internal architecture of the system under development is defined by SysML Internal Block Diagrams. For our example we have two alternatives for the FDS architecture as shown by Figures 7 and 8.

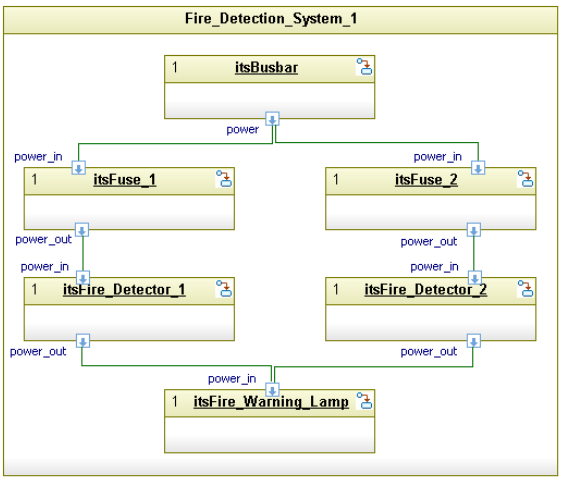

Figure 7: FDS System 1

Each of the components used in the implemenation of the FDS has a defined failure rate. The values used for the example calculation are given by Table 1.

5.4 Implementation and results

The concept was implemented using IBM Rational Rhapsody for creating the SysML models and a Java program, henceforth called SafetyAnalyzer. The SafetyAnalyzer uses the Rhapsody API to Rhapsody retrieve the models. Using the stereotypes defined in the system model the elements can converted into corresponding internal data types. The core

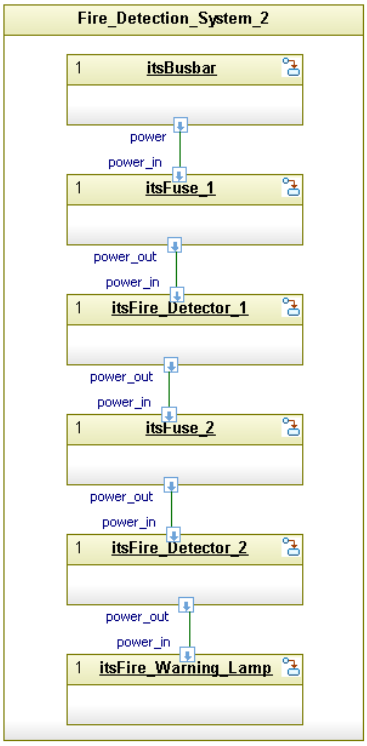

Figure 8: FDS System 2

Component	Failure rate
Busbar	4x10-6/h
Detector failed	5x10-5/h
Detector spurious	2x10-5/h
Fuse	1x10-6/h
Warning lamp	5x10-6/h

Table 1: Failure rates for FDS components

functionality uses this internal data model which is independent of the model implementation and therefore is restricted neither to the use of SysML nor to a particular tool such as Rhapsody.

The internal data model essentially consists of graphs: one for the functional architecture and one for each of the system implementations. Simply put, the safety calculations involve traversing the graph of the functional architecture while at the same time following the graph of the system architecture and gathering a set of visited component nodes. Then, the minimal cut set can be derived from this set of visited component nodes by applying the principles described in chapter 3. Given the minimal cut set, the failure rates of the components can be directly used to calculate a final number for the probability of the failure case to happen.

The SafetyAnalyzer provides as output:

- The minimal cut set for each failure case and system alternative.
- A reliability block diagram representing this minimal cut set graphically.
- A safety verdict that shows for each failure case and system alternative if this system alternative is able to stay within the bounds for the maximum allowed failure rate.
- An overall safety verdict that states if a system alternative is able to adhere to the restrictions by all defined failure cases.

For the FDS, for 1 hour of operation the results given by the automatic safety analysis are as follows:

- **FDS 1**
  - Loss of detection: Busbar + (Fuse 1 + Detector 1 failed) * (Fuse 2 + Detector 2 failed) + Lamp = 9x10-6. Verdict: **Pass**
  - False indication: Detector 1 spurious + Detector 2 spurious = 4x10-5. Verdict: **Fail**
  - Overall verdict: **Fail**
- **FDS 2**
  - Loss of detection: Busbar + Fuse 1 + Detector 1 failed + Fuse 2 + Detector 2 failed + Lamp = 1,11x10-5. Verdict: **Pass**
  - False indication: Detector 1 spurious * Detector 2 spurious = 4x10-25. Verdict: **Pass**
  - Overall verdict: **Pass**

Additionally our implementation provides the RBDs for each system implementation and failure case as a .gml[4] file. The RBDs produced for the FDS visualized using yEd[5] are shown by Figures 9 and 10.

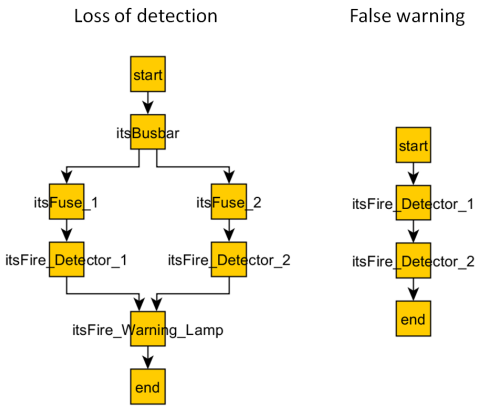

Figure 9: RBDs for FDS1

[4]Graphlet GML graph data format used for the storage and exchange of graphs. GML is an acronym derived from Graph Modelling Language.
[5]`http://www.yworks.com/en/products_yed_about.html`

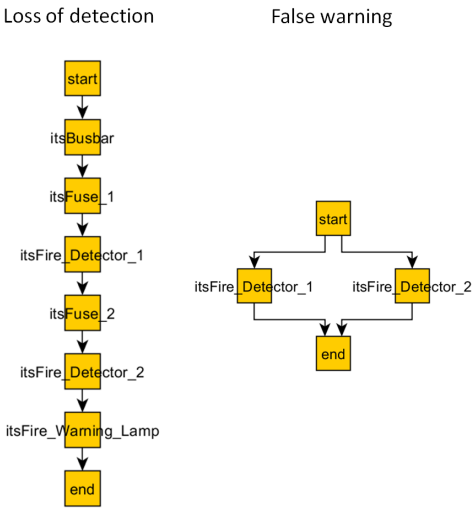

Figure 10: RBDs for FDS2

6. CONCLUSIONS

The MBSA process developed in this work was motivated by the idea of an automated, light-weight, function-oriented and statical safety analysis of architectures modeled in SysML in a single tool. In doing so, this MBSA process has a different intention than other existent safety analysis approaches which mostly work with a safety model of a system which is not identical to the design model of that system. From that it follows, that model-to-model transformations have to be applied between the systems engineering domain and the safety domain. The avoidance of this error-prone and time-intensive transformation of system models is the main benefit of this approach. For this, SysML was extended to include safety-related information. Since these extensions are realized by stereotypes, applying it to existing SysML system models requires almost no additional modeling effort. In doing so, the system designer can execute safety analyses on system designs without the help of safety engineers and thus gets a fast safety-related feedback of design decisions. However, this MBSA approach does not claim to be a substitute for a certification relevant safety analysis. For that purpose, complex safety analysis techniques have to be used which cover also dynamic aspects of the system. Nevertheless, the approach introduced in this work can be seen as a support for designing structurally safe systems on the basis of functional safety requirements.

7. REFERENCES

[1] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2):109–124, 1999.

[2] A. Birolini. *Reliability engineering: theory and practice*. Springer Verlag, 2010.

[3] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, et al. ESACS: an integrated methodology for design and safety analysis of complex systems. *ESREL 2003*, pages 237–245, 2003.

[4] P. David, V. Idasiak, and F. Kratz. Reliability study of complex physical systems using SysML. *Reliability Engineering & System Safety*, 95(4):431–450, 2010.

[5] P. Fenelon, J.A. McDermid, M. Nicolson, and D.J. Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.

[6] Object Management Group. OMG Systems Modeling Language (OMG SysML) Version 1.3. *OMG document: formal/2012-06-01*, 2012.

[7] C. Haskins, K. Forsberg, and International Council on Systems Engineering. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. 2011.

[8] M. Hecht, A. Lam, and C. Vogl. A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 361–366. IEEE, 2011.

[9] P. Helle, C. Strobel, A. Mitschke, W. Schamai, A. Rivière, and L. Vincent. Improving systems specifications a method proposal. CSEM, 2008.

[10] SAE International. *ARP 4754A - Certification considerations for highly-integrated or complex aircraft systems*. 1996.

[11] SAE International. *ARP 4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. 1996.

[12] R. Jacquart. *Building the Information Society: IFIP 18th World Computer Congress: Topical Sessions 22-27 August 2004, Toulouse, France*, volume 156. Springer, 2004.

[13] N.G. Leveson. Software safety: Why, what, and how. *ACM Computing Surveys (CSUR)*, 18(2):125–163, 1986.

[14] N.G. Leveson. *Safeware: system safety and computers*. ACM, 1995.

[15] E. Lloyd and W. Tye. *Systematic safety: safety assessment of aircraft systems*. Civil Aviation Authority, 1982.

[16] Ministry of Defence. *Standard 00-56 Issue 4-Safety Management Requirements for Defence Systems*. 2007.

[17] Y. Papadopoulos and J. A McDermid. The potential for a generic approach to certification of safety critical systems in the transportation sector. *Reliability engineering & system safety*, 63(1):47–66, 1999.

[18] D. Powell. Failure mode assumptions and assumption coverage. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 386–395. IEEE, 1992.

[19] M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J. Minarick, and J. Railsback. Fault tree handbook with aerospace applications, version 1.1. *National Aeronautics and Space Administration*, 2002.

[20] M.T. Todinov. *Risk-based reliability analysis and generic principles for risk reduction*. Elsevier Science, 2007.

[21] A.K. Verma, S. Ajit, and D.R. Karanki. *Reliability and Safety Engineering*. Springer Verlag, 2010.

Real–Time Design Models to RTOS-Specific Models Refinement Verification

Rania Mzid, Chokri Mraidha
CEA List, Laboratory of model driven engineering for embedded systems
Point Courrier 174, Gif-sur-Yvette, 91191, France
rania.mzid@cea.fr
chokri.mraidha@cea.fr

Jean-Philippe Babau
Lab-STICC, UBO, UEB
Brest, France
Jean-Philippe.Babau@univ-brest.fr

Mohamed Abid
CES Laboratory
National school of engineers of Sfax
Sfax, Tunisia
Mohamed.Abid@enis.rnu.tn

ABSTRACT
One key point of Real-Time Embedded Systems development is to ensure that functional and non-functional properties (NFPs) are satisfied by the implementation. For early detection of errors, the verification of NFPs is realized at the design level. Then the design model is implemented on a Real-Time Operating System (RTOS). However, the design model could be not implementable on the target RTOS. In this paper, we propose to integrate between the design and the implementation phases, a feasibility tests step to verify whether the design model is implementable on the target RTOS and a mapping step to generate the appropriate RTOS-specific model. This two-steps approach is based on an explicit description of the platform used for verification and the RTOS which is the implementation platform. Moreover an additional verification step is needed to ensure the conformity of the implementation model to the design model with regard to NFPs.

Keywords
MDD, Design Model, RTOS-specific Model, Real-Time Validation;

1. INTRODUCTION
In order to overcome the increasing complexity of Real-Time Embedded Systems (RTES), Model-Driven-Development (MDD) [1] promotes a rise in level of abstraction by introducing intermediate models from specification to implementation, while passing through design, and enabling validation at each level.

At the design level, scheduling analysis [2] may be applied to validate design choices in terms of timing requirements. Several tools are available to carry out such validation, one can cite as example Qompass-Architect [3], Cheddar [4], and MAST [5]. However, to achieve that, each of these tools considers some implementation assumptions (e.g. scheduling policy, communication mechanisms) which are related to a *validation platform*. On the other hand, there is an important number of Real-Time-Operating-System *(RTOS)* in the market. Some are compliant to a specific standard such as POSIX [6], OSEK-VDX [7] and μITRON [8], some are commercial and others are free and may be not compliant to any standard. These RTOS or standards share common concepts but with specific features [9]. The choice of the target RTOS depends on the considered community and the intended use [10].

From these considerations, the refinement of the design model, making some implementation assumptions to be validated, to an RTOS-specific model is error-prone. In fact, the selected RTOS may be too restrictive with regard to the validation platform or incoherent correspondence between properties of the validation platform resources and the RTOS ones may occur. In that case, the designer iterates on the design model, modifying and re-validating it, looking for an *implementable* solution. These modifications are usually based on the designer experience and reduce portability of design model: the design model becomes specific to an RTOS.

Several works are interested in the deployment of an application on a real platform. In [11], the authors propose a generative process to transform an application deployed on one RTOS to another based on an explicit description of the latter using the Software Resource Modeling (SRM) UML profile, which is part of the UML profile for MARTE [12] . This work makes the assumption that the deployment is always possible and did not paid any attention to the incoherence between the characteristics of the different platform resources and its influence on the validity of the obtained model. The author in [13] proposes a deployment process of an application on a RTOS. This process considers also an explicit description of the latter but using a Domain Specific Language (DSL) called RTEPML and focuses on defining generic transformations to automate the process. Compared to [11], this approach claims the necessity to verify the availability of a concept on the target RTOS before the deployment.

In previous work [14], we proposed a two-steps approach that ensures the generation of valid implementation model from design model fulfilling timing properties. This approach is based on an explicit description of the validation platform and the target RTOS using SRM [12]. The first step, which is a set of feasibility tests, aims at verifying whether the implementation assumptions made at the design level are implementable on the target RTOS. The second one is a mapping step that performs the mapping between the validation platform resources and the RTOS resources to obtain a RTOS–specific model. In [14], we have focused on concurrency aspects, scheduling policies, tasks and we have especially treated the tasks' priority problem. In this paper, we extend the proposed approach by treating the shared resources aspect. Thus, we consider that tasks in the design model may be dependent by sharing resources and we describe the required resources in the validation and RTOS platforms. We discuss also the additional feasibility tests and mapping necessary from this perspective. On the other hand, one important issue is to verify whether the generated RTOS-specific model is valid with respect to the design model. So, in this paper, we focus also on identifying the required verification to confirm the correctness of such model.

The paper is organized as follows. Section 2 presents the assumptions of the paper. In section 3, we give an overview of the two-steps approach to generate valid implementation models and we add the necessary treatments for the shared resources

aspect. In section 4, we explain how to verify the validity of the RTOS-specific model with respect to the design model. Section 5 illustrates on an example the approach and the verification phase. Finally, section 6 concludes the paper.

2. ASSUMPTIONS

We assume that timing validation is performed at the design level using Optimum methodology [3] supported by the Qompass-Architect tool. This methodology introduces timing validation from the specification level in order to guide the design of the concurrency model that satisfies the timing constraints.

In this paper, we assume that the design model, generated by Optimum consists of a set of tasks $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ executing the different functions of the system. All tasks in the model are scheduled according to their priority. So each task τ_i is characterized by its priority P_i and runs at a base period T_i. Besides, we assume that two tasks in the model may be dependent by sharing resources. Consequently, the design model consists also of a set of resources $R = \{r_1, r_2, \ldots, r_m\}$ such as each $r_i \in R$ is shared between two tasks or more.

Finally, we suppose that the hardware architecture corresponds to a single execution node (mono-processor architecture).

From this correct model (design model), one objective of this work is to ensure a correct transition to the implementation model while respecting the timing properties. More precisely, we focus on platform aspect because validation is based on a validation platform, here the platform used by Qompass-Architect, while implementation is based on the RTOS.

3. MODEL-DRIVEN APPROACH

In this section we give an overview of the two-steps model-driven approach which has been explained in details in previous work [14]. Then, we extend this approach by considering shared resources aspect.

3.1 Overview

One key point of our approach is to ensure a correct deployment of a design model satisfying non–functional requirements on an RTOS. The obtained RTOS-specific model (implementation model) must conserve the properties that have been validated at the design level.

The design model translates the system specification and fulfills its timing constraints under the assumptions made by the validation platform related to the validation tool (c.f. Figure 1). In our case, the validation platform is the Optimum platform as we assume that timing validation is performed at the design level using Qompass-Architect [3]. In fact, the validation platform includes all concepts provided by RTOSs and that are necessary to perform timing validation. This makes this platform independent from a particular RTOS and provides a flexible framework to the designer to make different design choices.

In our approach, we choose an explicit description of the validation platform and the RTOS using SRM. Indeed, SRM allows capturing the semantics of the different concepts defined in both platform models and serves as a pivot language to automate the refinement of the design model to an implementation model.

As shown in Figure 1, the approach introduces two steps between the design and the implementation levels:

- *Feasibility tests step:* this step generates an *error* when the design model is not implementable on the target RTOS and provides a feedbacks to the designer to inform him about the source of the problem. It generates a *warning* when the design model is implementable but the RTOS provides an implementation that is probably more optimized than the one chosen at the design level. Otherwise, this step mentions that there is *no problem* and that the mapping step can be performed.

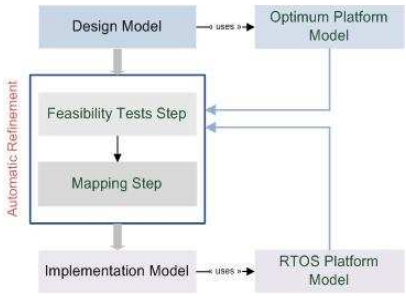

Figure 1.Model-Driven approach

- *Mapping step:* this step generates the RTOS-specific model by performing the mapping of concepts and the mapping of properties of these concepts. This mapping is based on the notion of matching between the resources of the validation platform and the RTOS one. This matching is ensured, in our case, by the use of SRM to describe both platforms (c.f. Figure 2).

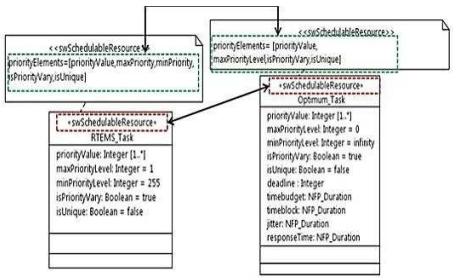

Figure 2.Matching using SRM

In [14], we were interested in concurrency aspects such as scheduling policies, tasks and their properties that describe the application behavior in a design model with *independent tasks*. The greater emphasis was on the priority problem. Briefly, we describe the tests invoked by the feasibility tests step which are related to the priority aspect.

- *Test of scheduler:* this test verifies the scheduling policies adequacy between the validation platform and the RTOS. For instance, if the scheduling policy used at the design level is priority-based and the RTOS does not offers a priority-based policy. So in that case, this test generates an error to mention that the input design model is not implementable on the target RTOS
- *Test of number of priority levels:* this test computes the number of priority levels used in the design model and verifies whether the platform supports that number. If the number of priority levels allowed by the RTOS is lower than the number used at the design level, this test generates an error to indicate that the design model is not implementable on the target RTOS.
- *Test of equal priority levels:* this test verifies if, at the design level, there are tasks that share the same priority levels. In that case, if the target RTOS does not support such situation, this test generates an error to inform the designer that his design model is not implementable.

In order to generate the RTOS-specific model, the mapping step provides also different mapping strategies of the priority values to give a flexible framework to the designer. We give also a brief description of these mapping strategies:

- *Direct mapping* keeps at the implementation the same priority values used in the design model. This type of mapping does not ensure always valid implementation models.
- *Linear mapping* generates consecutive values from the available minimum priority level of the used RTOS. If feasible, it ensures always valid implementation models. However, the generated priority is less convenient to insert new task at run-time.
- *Mapping by step* is similar to the previous one, but adding a step between two consecutive levels of priority. The validity of the obtained implementation model depends on the step size. Like for direct mapping, it is necessary to add a supplementary test to verify whether this mapping is possible.
- *Proportional mapping* distributes applicative priority values over the maximal range offered by the RTOS. It guarantees valid implementation models. Nevertheless, this type of mapping is not possible if the RTOS does not provide an upper bound of priority levels.

3.2 Consideration of shared resources

We suppose that tasks in the design model may be dependent by sharing resources (c.f. section 2). The sharing of a data resource, when the use of the data must be atomic, necessitates choosing three architectural parameters: the *synchronization protocol*, the *allocation policy* and the *access protocol*. The synchronization primitive (e.g. Semaphore, Mutex) is needed to ensure that one and only one task can use the resource at a time. The allocation policy or the waiting queue policy (e.g. FIFO, priority-based) determines what happens when a request is made for the resource when the resource is busy. Finally, the access protocol (e.g. PCP, PIP) is used to avoid priority inversion situations or deadlock by modifying the priority of the task during the execution. The combined choice of synchronization protocol, allocation policy and access protocol corresponds to a *possible implementation* of the shared resource (critical section). In next subsections, validation and RTOS platform models are enriched to support the creation of design and implementation models with shared resources. Then, we discuss the feasibility test and mapping steps from this perspective.

3.2.1 Validation platform model for Optimum

In order to perform timing validation, the designer has to make implementation assumptions on how to implement the critical section. As already discussed in section 3.1, the validation platform is an "ideal" platform that offers unlimited design choices for the designer and is independent from a particular RTOS. Consequently, this platform covers all the ways for implementing a shared resource. To this end, we add a *Shared_Resource* concept to the Optimum platform (c.f. Figure 3) and we annotate the latter with *"swMutualExclusionResource"* stereotype from SRM. The choice related on how to implement this shared resource corresponds to setting the values of the *mechanism*, *waitingQueuePolicy* and *concurrentAccessProtocol* properties of the *"swMutualExclusionResource"* stereotype which correspond respectively to the *synchronization protocol*, *allocation policy* and *access protocol* parameters already mentioned. In Figure 3, we choose a default implementation of the shared resource (*PCP_Semaphore*). However, the designer can change this implementation by modifying the values of these properties. Depending on the designer choices, the validation tool involves the corresponding analysis test.

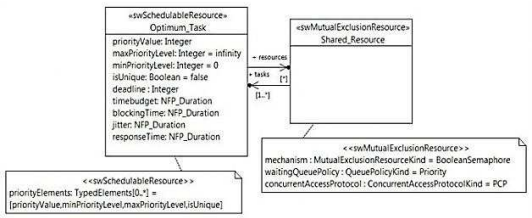

Figure 3.Excerpt of the Optimum platform model

Some combined choices of these three parameters do not correspond to real implementations. An example of non-meaningful implementation is; we choose a semaphore mechanism with a FIFO waiting queue and a PCP protocol. In order to avoid such situation, we propose to add an OCL constraint [15] for each non-meaningful implementation. The current implementation of SRM imposes to express these choices at the profile level (i.e. set the properties values of the *"swMutualExclusionResource"* stereotype). Consequently, the OCL constraints that prevent insignificant implementations of the critical section are also defined at the profile level. For instance, the previous unsound situation corresponds to a constraint associated to *"swMutualExclusionResource"* stereotype from SRM and is given just below:

```
context swMutualExclusionResource
inv:
(Self.mechanism = BooleanSemaphore) and
(Self.waitingQueuePolicy = FIFO) implies (not
(Self.concurrentAccessProtocol = PCP)
```

3.2.2 RTOS Model

To tackle the issue of shared resources, the RTOS model should describe the possible implementations of critical section provided by the considered RTOS. We choose, in this paper, RTEMS [16] as a target RTOS and we give in Figure 4 an excerpt (a view for the shared resources) of the RTEMS platform model.

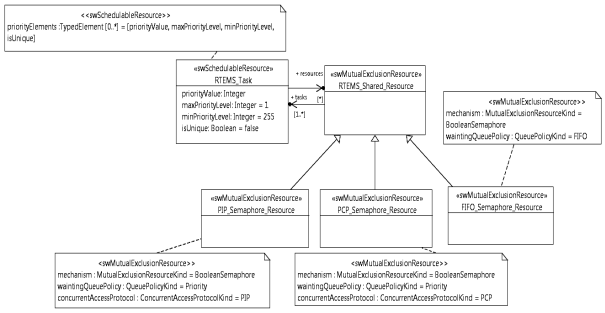

Figure 4.Excerpt of the RTEMS model

RTEMS provides three possible implementations of a shared resource. Each of these implementations corresponds to a class in the RTEMS model annotated *"swMutualExclusionResource"*. For each class, we give default values to the *mechanism*, *waitingQueuePolicy* and *concurrentAccessProtocol* properties of the *"swMutualExclusionResource"* stereotype which define the considered implementation. For example, The *FIFO_Semaphore_Resource* concept corresponds to a shared resource implementation using a Boolean semaphore as a

synchronization protocol and FIFO as an allocation policy. This implementation should not define an access protocol this is why it does not appear in Figure 4.

3.2.3 Feasibility tests and mapping steps

The extension of our approach to support tasks dependencies by sharing resources requires additional feasibility test to verify whether the target RTOS provides the critical section implementation chosen at the design level. If it is not the case, the feasibility tests step generates *an error* to inform the designer that the corresponding design model is not implementable on this RTOS.

In some cases, the implementation choices made by the designer to implement the critical section are implementable; however the RTOS provides another implementation that offers better real-time performance. In that case, the feasibility tests step generates *a warning* in order to propose to the designer to change the implementation. An example of such situation is; when the designer chooses a FIFO semaphore at the design level and the target RTOS provides a PIP semaphore. So the feasibility tests step highlights a warning to inform the designer that the target RTOS provides a PIP semaphore which is more adapted for real-time application [17]. The designer at this point can choose to keep his design model and to perform the mapping or to modify the implementation choices for the critical section taking into consideration the generated warning.

The mapping step for the shared resources is straightforward. If the design model is implementable, this step creates an instance at the implementation level of the resource that defines the same critical section implementation choices made at the design level.

4.RTOS-SPECIFIC MODELVERIFICATION

One key point of the proposed approach is to generate correct RTOS-specific models from valid real-time design models. In order to confirm that the obtained model is correct (i.e. preserves design model timing properties); some properties must be verified at the implementation level. In our case, we identify three properties:

- **P1**: the priority values of the different tasks must be always within the range of priority values allowed by the RTOS
- **P2**: the execution order of the different tasks defined at the design level must be preserved at the implementation level.
- **P3**: the access order to shared resources must be preserved

To address the first property (**P1**), we propose to add an OCL constraint to the RTOS model. The role of this constraint is to verify that the priority values of the different tasks in RTOS-specific model are meaningful to the considered RTOS. As an example, we give just below the constraint that we add to the RTEMS model and which corresponds to (**P1**):

```
context RTEMS_Task
inv:
if(Self.minPriorityLevel< Self.maxPriorityLevel) then
Self.minPriorityLevel < Self.PriorityValue <
Self.maxPriorityLevel
else
Self.maxPriorityLevel< Self.PriorityValue <
Self.minPriorityLevel
endif
```

Depending on the priority order (increasing or decreasing) which is determined by the *minPriorityLevel* and *maxPriorityLevel* attributes of the *RTEMS_Task* (c.f. Figure 4), this constraint verifies whether the priority values of the different tasks instances of the *RTEMS-task* are between the minimum and the maximum priority levels (given by *minPriorityLevel* and *maxPriorityLevel* and correspond respectively to 255 and 1 in RTEMS).

For the second property (**P2**), we don't focus on the priority values (which is already verified by the first property) but on the execution order of the different tasks which must be equivalent at the design and implementation level. In order to verify this property, we propose the meta-model given in Figure 4.

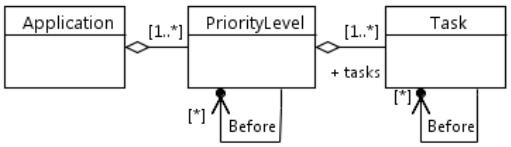

Figure 4.Verfication-oriented meta-model

This meta-model considers an application as a relation of precedence among priority levels. As we may have tasks that share same priority level, we consider in the meta-model that at each priority level one or more tasks may also have a relation of precedence. So this meta-model considers that the most important is not the values of priority but the relation of precedence between them. In order to verify the second property, we transform the design and the RTOS-specific models to models that conform to this meta-model and we verify if they are equivalent.

The access order to the shared resource in the model is preserved at the implementation level, if and only if, the execution order of tasks that share this resource is also preserved. Consequently, the third property (**P3**) is verified, if and only if, the second property (**P2**) is verified.

As a conclusion, the generated RTOS-specific model is correct with respect to the design model, if and only if, these three properties (**P1**), (**P2**) and (**P3**) are verified.

5. CASE STUDY

In this section we illustrate our approach with the example presented in [3]. This example corresponds to a classical case study in the automotive domain, i.e. the antilock control sub system.

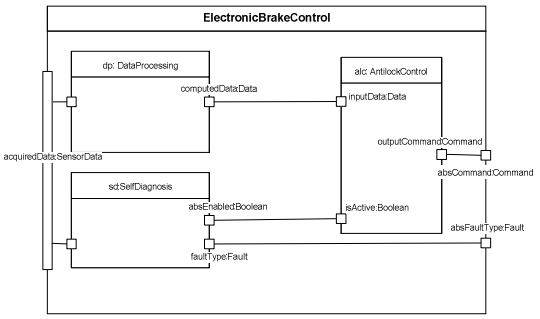

Figure5. System functional model

This subsystem is a classic sensor-controller-actuator system. Figure 5 gives a structural view of the main functions inside the controller: a data processing function for data coming from the sensor, the anti-locking brake function calculating the command

to send to the actuator, and a diagnosis function that disables the anti-locking function in case a fault in the subsystem is detected.

Each function above has an associated behavior modeled here as an activity. Figure 6 below complements the structural functional model with the description of the system end-to-end scenarios. Two events (*acquisitionForAbs* and *acquisitionForDiagnosis*) are triggering the sequences of functions behavior execution.

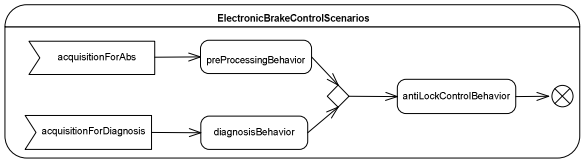

Figure 6 .System-level behavior

From this behavioral description, Qompass-Architect generates the design model given in the next subsection. This generation relies also on some additional parameters defined in the system specification such as the activation periods of events and time budget of actions.

5.1 Design model

Figure 7 gives a schematic view of the design model of the antilock control subsystem generated by Qompass-Architect.

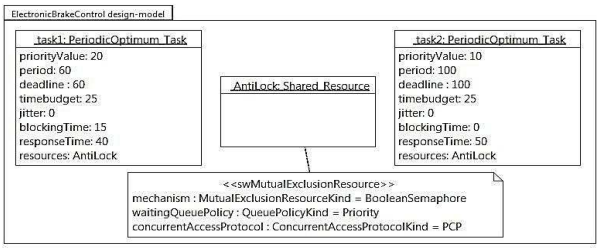

Figure 7.Design model

This model consists of two periodic tasks *taks1* and *task2* which are instances of the *PeriodicOptimum_Task* concept of the Optimum platform. The first task, *task1,* is triggered by the event *acquisitionForAbs;* consequently its period corresponds to the period of this event (*60 ms*). Besides, this task executes *preProcessingBehavior* and *antiLockControlBehavior* actions and then its execution time (*timebudget*) is the sum of the execution times of these two actions. Similarly, *task2* is triggered by the *acquisitionForDiagnosis* event and executes the *diagnosisBehavior* and *antiLockControlBehavior* actions. Qompass-Architect gives to *task1* a priority value equals to 20 and to *task2* a priority value equals to 10. The priority order in the Optimum platform is increasing as specified in Figure 3 (given by *minPriorityLevel* and *maxPriorityLevel* attributes of the *Optimum_Task*). Accordingly, *task1* has a higher priority than *task2*. These two tasks are dependent by sharing the *AntiLock* resource which corresponds to the *antiLockControlBehavior* action. Qompass-Architect chooses to implement this critical section with a PCP_Semaphore (i.e. Boolean semaphore as a mechanism, a priority-based as a waiting queue and PCP as an access protocol).

Based on these different implementations choices (priority assignment, critical section implementation, tasks number…), Timing validation is performed to verify whether this design model meets its timing requirements. Indeed, Qompass-Architect computes the blocking time depending on the implementation choices of the critical section and then computes the response time of the different tasks. The result of this validation is also given in Figure 7. From this figure, we can conclude that this model is valid from a real-time perspective since the response times of *task1* and *task2* are lower than their deadlines.

We aim at generating a correct RTEMS-specific model from this valid design model. This implementation model must conserve the timing properties while considering the characteristics of the RTEMS platform. In the following subsection, we give this RTEMS-specific model.

5.2 RTEMS-specific model

The generation of the RTEMS-specific model following our approach requires passing through two steps; feasibility tests and mapping.

For this design model, the feasibility tests step involves the different feasibility tests related to the priority aspect (c.f. section 3.1) and the shared resources aspect (c.f. section 3.2.3). For this design model, this step does not raise any feasibility concern: the design model is implementable on RTEMS. Consequently, we process the mapping step to generate the RTEMS-specific model. Figure 8 gives a schematic view of the RTEMS-specific model.

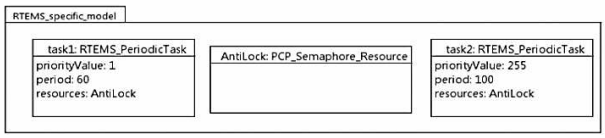

Figure 8.RTEMS-specific model

This step performs the mapping between Optimum platform resources and RTEMS resources; and the properties of these resources. Consequently, RTEMS-specific model consists of two tasks instances of *RTEMS_PeriodicTask* concept which corresponds to the appropriate type that matches the *PeriodicOptimum_Task* resource in Optimum platform. It consists also of an *AntiLock* resource instance of the *PCP_Semaphore_Resource* which defines the same implementation of critical section chosen in the design model.

For the properties, the mapping step detects that the only three properties that require mapping are the *priorityValue*, the *period* and *resources* (the other properties have a default value or they are not referenced in both platform models). This step proposes different strategies to perform the mapping of priority values (c.f. section 3.1). In Figure 8, we choose the proportional mapping of the priority values. The period values are expressed in *ticks* in RTEMS and the duration of a *tick* is configurable. We suppose here that the *1 tick* is equal to *1 ms*. This is why; the values of the period are kept the same at the implementation model. Finally, for the resource property we perform a direct mapping to keep the information that this resource (*AntiLock*) is shared between *task1* and *task2*.

5.3 RTEMS-specific model verification

In order to verify the correctness of the generated RTEMS-specific model, three properties already explained in section 4 should be fulfilled.

The first property **(P1)** is that the priority values of the different tasks in the implementation model are between the minimum and maximum priority levels allowed by the RTOS which correspond respectively to 1 and 255 for RTEMS. We can conclude from Figure 8 that this property is verified as the priority values of *task1* and *task2* are within this interval.

The second property (**P2**) is that the execution order of the different tasks is equivalent at the design and the implementation level. To this end, we transform the design model to a model instance of the verification-oriented meta-model given in Figure 4. This model (c.f. Figure 9) considers that our application is a relation of precedence among two priority levels LD1 and LD2. Each priority level references one or several tasks from the design level. In our case, we have just one task for each level since the design model does not define tasks with equal priority levels. From this model, the most important information is that, at the design level, *task1* is executed before *task2*.

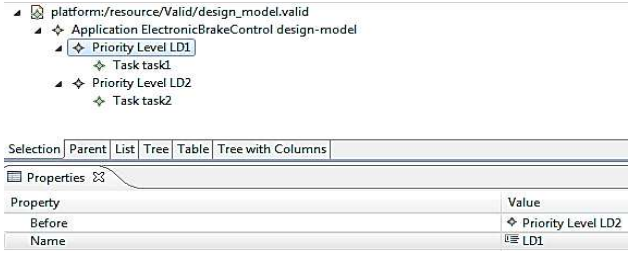

Figure 9.Design model as an instance of the verification-oriented meta-model

In the same way, we transform the RTEMS-specific model to a model conforming to the verification-oriented meta-model given in Figure 4. This model is given in Figure 10 and defines also a relation of precedence among two priority levels LI1 and LI2. Each level references also one task from the implementation model.

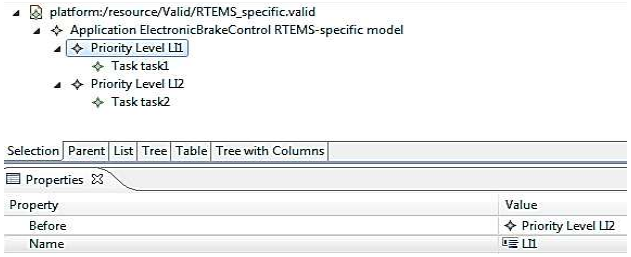

Figure 10.RTEMS-specific model as an instance of the verification-oriented meta-model

From Figure 9 and Figure 10, we conclude that the second property is also fulfilled. In fact, even the priority values at the design and the implementation levels are different; the execution order of the two tasks is conserved.
The third property (**P3**) is verified since the second property is fulfilled.
All the properties are verified and thus the generated RTEMS-specific model is correct.

5. CONCLUSION

In this paper, we propose an approach to ensure an automatic correct transition from a valid design model to an RTOS-specific model that conserves timing properties. This approach is based on two steps; the first step verifies whether the design choices are implementable on the target RTOS and the second step perform an appropriate mapping to generate the RTOS-specific model. In order to assess that the obtained RTOS-specific model is correct with respect to the design model, we identify the properties that should be verified and we propose a way to check them at the implementation level.

As future work, we aim at considering other aspects such as activation patterns, communications in a distributed platform. For each aspect, we define the additional feasibility tests and mapping strategies. Another perspective consists in refactoring the design model, when the latter is not implementable, based on the feasibility tests step feedbacks.

REFERENCES

[1] B. Schtz, A. Pretschner, F. Huber, J. Philipps. Model based development of embedded systems, Lecture Notes in Computer Science, vol 2426, 2002, Springer, 2002, pp.331-336.

[2] L. Sha, T. Abdelzaher, , K. E. Arzen., A. Cervin, T. P. Baker, A. Burns , G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. Real-Time Systems 28(2/3): 101155. 2004.

[3] C. Mraidha, S. Tucci Piergiovanni and S. Gerard: Optimum: a MARTE-based methodology for schedulability analysis at early design stages. ACM SIGSOFT Software Engineering Notes 36(1): 1-8 (2011)

[4] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a Flexible Real Time Scheduling Framework. International ACM SIGADA Conference, Atlanta, November 2004.

[5] M. GonzAlez Harbour, J.J. GutiCrrez Garcia, J.C. Palencia GutiCrrez, and J.M. Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. Real-Time Systems, 13th International Euromicro Conference. Delft, June 2001.

[6] The Open Group Base Specifications, Portable Operating System Interface (POSIX), ANSI/IEEE Std 1003.1, 2004.

[7] OSEK Group. OSEK/VDX Operating System Specification. http://www.osek-vdx.org.

[8] T-Engine Forum. µITRON 4.0 Specification, July 2010. http://www.t-engine.org

[9] R. Yemhalli. Real-time operating systems: An ongoing review. In Work-ln-Progress Sessions. The 2lrl IEEE Real-lime System Symposium (RTSSWIPOO), Orlando, Florida, November 2000.

[10] H. Takada, Y. Nakamoto, and K. Tamaru, "The ITRON Project: Overview and Recent Results", 5th International Conference on Real-Time Computing Systems and Applications (RTCSA), pp.3-10, Oct. 1998.

[11] F. Thomas, J. Delatour, F. Terrier, and S. Gerard. Toward a framework for explicit platform-based transformations. In Proceeding of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC). Orlondo, Florida, USA, May 2008.

[12] Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Object Management Group, Inc., September 2010, OMG document number: ptc/2010-08-32

[13] M. Brun. Contribution à la considération explicite des plates-formes d'exécution logicielles lors d'un processus de déploiement d'application. PhD Thesis university of Nantes. October 2010.

[14] R. Mzid, Ch. Mraidha, J-P. Babau, M. Abid. A MDD Approach for RTOS Integration on Valid Real-Time Design Model. The 38th Euromicro Conference On software Engineering and Advanced Applications (SEAA'12), Cesme, Izmir, Turkey, September 2012.

[15] Object Management Group, Object Constraint Language (OCL). Object Management Group, Inc., May 2006, OMG document number: formal/06-05-01

[16] RTEMS C Users Guide. Edition 4.6.5, for RTEMS 4.6.5. August 2003.

[17] Mark H. Klein, Th. Ralya, B.Pollak, R. Obenza and M.Gonzalez Harbour. A Practitioner's Handbook for real-Time Analysis. Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publisher. ISBN 0-7923-9361-9. p. 5-30.

Component-Based Models for Runtime Control and Monitoring of Embedded Systems

Tobias Schwalb
schwalb@kit.edu

Tobias Gädeke
gaedeke@kit.edu

Johannes Schmid
schimd@kit.edu

Klaus D. Müller-Glaser
mueller-glaser@kit.edu

Karlsruhe Institute of Technology
Institute for Information Processing Technologies
Engesserstr. 5, 76131 Karlsruhe, Germany

ABSTRACT

Nowadays, more and more developments in the embedded systems domain are based on components and abstract models. However, while the design becomes more abstract, control and monitoring during runtime are often performed on low abstraction levels. In contrast to this low level access we present a seamless design flow for adjustment and error identification using abstract component-based models. We develop an extended metamodel to describe components and their platforms and the connection between the model and the real hardware. Furthermore, we integrate on model level platform abilities for control and especially debugging to support for example real-time recording. From a user's perspective the system is designed, controlled and monitored on model level. We discuss different methods concerning runtime control and monitoring of resource constraint systems. We demonstrate the concept's applicability based on two exemplary use cases: wireless sensor network application engineering and reconfigurable hardware development.

1. INTRODUCTION

The complexity in embedded systems increases nowadays, due to the rising functionality, demands and shorter product cycles. Further challenges arise due to parallel and distributed systems in connection with limited computing power, interfaces and debugging capabilities. To handle this rising complexity, more and more model-based design methods are used [10]. These methods aim at enhancing abstraction, scalability, maintainability and interoperability to increase quality and decrease development time.

One methodology is to use component models. A component encapsulates a set of related functionality and data. Components communicate with each other via interfaces and are configured using parameters. The main advantages of components are their abstraction and the possibility of hierarchical reuse in other developments. In comparison to present concepts (see Section 2) we concentrate on component-based models for runtime adjustment and error identification and consider the abilities, limitations and dependabilities in embedded systems [6]. Especially, we examine the abilities of on-chip control and debugging and integrate them on model level. Our goal is to allow abstract runtime control and monitoring based on the same component model used at design time. Therefore, we extend our previous work [13]. We constructed a new metamodel to be able to describe components in more detail and also reflect the abilities of the platforms as well as the connection between model and hardware. We also integrate recording and debugging scenarios executed central or on a target platform, which get managed from model level. Additionally, we evaluated our approach with two diverging application examples.

2. RELATED WORK

In industry, e.g. MATLAB / Simulink allows the design of systems using predefined building blocks and can generate C-code for embedded devices. During runtime the model can be connected to hardware to monitor functional reactions. This, however, can only be done for specialized devices. Lettner et al. discuss to use Rhapsody in the development of embedded systems [10]. In terms of the runtime connection, they describe the problem that graphical debugging with animations is too slow for practical usage. The design and usage of further runtime models and corresponding tools are described in [15] and [8].

Component models are widely used in the software domain (e.g. .NET, CORBA and EJB). These models are, however, generally complex and heavyweight and introduce large overheads. The component-based approach for small and large embedded systems is discussed in [2]. Further domain specific languages (DSL) are introduced which are partly based on the above languages. Cadena [7] is an IDE for component-based distributed embedded systems. It concentrates mainly on static methods to analyze systems. In [16] the Virginia Embedded Systems Toolkit (VEST) is introduced, which focuses on the evaluation of different constraints. It also considers real-time applications, but does not focus on functionality debugging. A platform independent component modeling language is shown in [1]. It supports distributed as well as real-time embedded systems and uses a component middleware. In terms of runtime activities it considers the deployment and target environment. A Unified Modeling Language (UML) profile is the Modeling and Analysis of Real-Time and Embedded systems (MARTE). It is used in the design of a complex heterogeneous component-based system [9]. The authors focus on the design, but do not consider runtime aspects. Rich Component Models for enhancing reuse in embedded systems are shown in [3]. These are used as a uniform representation of different de-

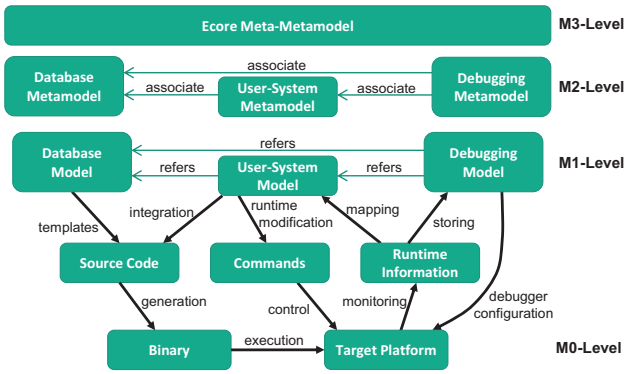

Figure 1: Concept for model-based design, control and monitoring of embedded systems.

sign entities to support management of functional and non-functional parts in the development. A framework for design and runtime debugging of component-based systems is shown in [17]. It enables to propagate checks from the specifications to application code to validate interactions between components. A framework and DSL for management of the communication between heterogeneous resource-constrained devices is shown in the ThingML-project [5].

In comparison to present concepts, we combine functional control and monitoring of white-box elements with abstract component-based architectural models. We explicitly model control and monitoring of components and consider platform specific functionalities to interact with embedded systems from model level.

3. CONCEPT

Our concept follows the Model Driven Architecture (MDA) introduced by the Object Management Group (OMG). We focus as part of the MDA on the Meta Object Facility (MOF) to describe our meta data architecture [11]. In addition, we use adapted component models, based on the UML.

Our concept in relation to the different levels of the MOF is depicted in Figure 1. On the M3-Level we use the *Ecore Meta-Metamodel*, because our implemented model-based development environment (see Section 6) is built using the Eclipse Platform. We developed a metamodel for the M2-Level, which extends existing component-based models and composes of three relating parts. The first part describes a *Database*, which stores individual components and platforms as well as their abilities, characteristics and dependabilities. Furthermore, it includes the connection between the model level representation and the hardware implementation. The modular elements are designed for easy reuse, assembling and setup of the actual user-system. The *User-System Metamodel* describes this actual user-system and is build upon these elements. Thereby, the platforms and components are connected and configured. The model is used as a mutual representation at model level of the hardware system at design time and runtime. In this relation, the *Debugging Metamodel* manages the setups for debugging in the IDE and on the platforms, which are described on model level. It also holds recorded data for playback and analyzing.

The M1-Level is the level the developer is working on. In the *Database Model*, as instances of the classes in the metamodel, the individual components and platforms are specified. In the *User-System Model* on M1-Level the defined components and platforms are instantiated, connected to each other and configured to form the actual user-system. Based on this and predefined component and platform templates the *Source Code* for implementation is generated. This usually is a high level code, like C++ or VHDL. To enable runtime control and monitoring, this code needs to include communication and debugging capabilities, if not directly supported by the embedded *Target Platform* (see Section 5). Using this code, a platform specific IDE generates a *Binary* on M0-Level to program the embedded target.

After programming the *Target Platform* can be controlled by modifying the *User-System Model* on M1-Level at runtime. According to changes of the user and information in the *Database Model* the IDE generates and sends *Commands* to control the *Target Platform*. Thereby, restrictions apply as not every element can be changed during runtime. The *Debugging Model* manages debugging and recording. It controls debugging in the IDE as well as on the *Target Platform* from model level. The monitored *Runtime Information* is interpreted and displayed in the *User-System Model* at M1-Level and stored in the *Debugging Model*.

4. CONNECTIVE METAMODEL

As introduced in Section 3, the metamodel is built on three relating parts according to a database of components and platforms, a model to describe the actual user-system and a part for managing debugging. Within the metamodel different constraints apply, which are not included in the metamodel to decrease its complexity. The constraints are specified using the Object Constraint Language (OCL) or, alternatively, are directly implemented in the IDE (Section 6).

Figure 2 depicts the metamodel without the attributes of the classes. The *Database* part is shown on the left side, it is adapted and extended from platform independent component metamodels in software domain. It describes *Components* with *Interfaces*, *Parameters* and *Limitations*. Thereby, a component might have different hardware *Implementations* to be optimized, e.g., for functionality, available parameters or runtime abilities. An *InOut* interface models the bus connections in the embedded system. According to the *Parameters*, we distinguish between three different types in an enumeration to differ between design time and runtime data as well as adjustability. *Configuration* parameters can be changed during the design time only, as they adapt generated source code. *Control* parameters can be modified during the design time and at runtime and set the value of variables or signals. *Monitoring* parameters are read-only for the user and display a status during runtime. A parameter can be either a *ParameterNumber* to store a numerical value or a *ParameterList* which excepts values from a predefined list only. Both types provide a limitation of the value range and are therefore suitable in the embedded domain for direct translation into hardware. To form this connection to the hardware the *ImplementationCoding* class is used in different aggregations of parameters and their possible values. Using the *available* association, the availability of the coding (and also of the connected element) can be specified depend-

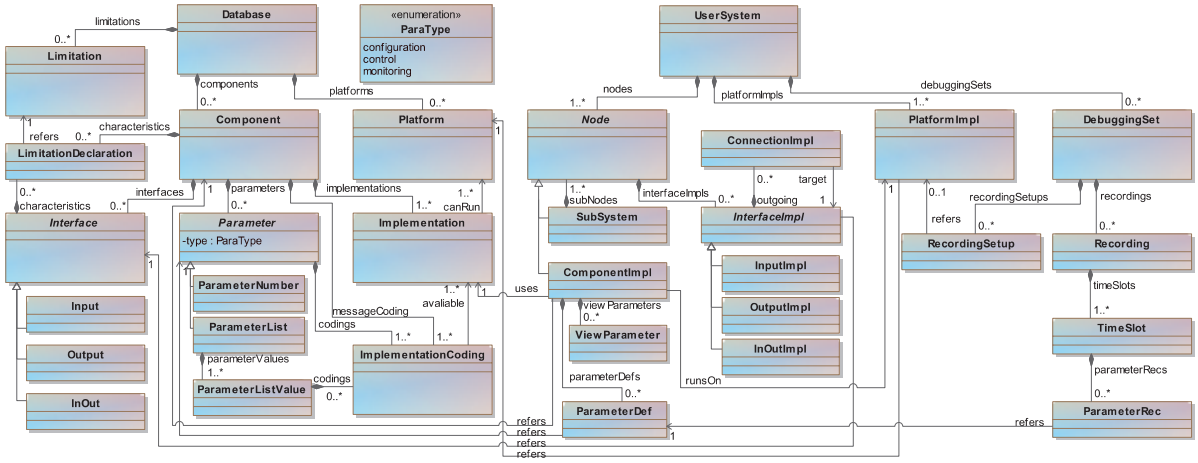

Figure 2: Metamodel for Database, User-System and Debugging.

ing on the implementation. Further limitations to component relations and interface connections are integrated using *provide* or *require* annotations in the *LimitationDeclaration* class. In addition, the database includes information about target *platforms* and their abilities.

The second part of the metamodel, depicted in Figure 2 in the middle, describes the actual *UserSystem*. Using the *Node* and *Subsystem* structure it can be built hierarchically. The *PlatformImpl* and *ComponentImpl* are implementations of the corresponding elements in the database. When the user integrates a component in the user-system, it gets automatically instantiated with its *InterfaceImpl* and *ParameterImpl* with references to the respective objects in the database. The *ConnectionImpl* connects the *InterfaceImpl* to form a network. The components are configured by setting the values of their parameters in the *ParameterDef* class. The IDE handles the limitations to ensure valid connections and parameter values. The class *ViewParameter* handles special parameters, which are used to specify relations to graphical representations. Therefore, the graphical representation can adapt, also during runtime, to the status of the component. Through these connection and limitations between database and user-system different scenarios can be constructed through reuse. In addition, the abstraction keeps easy (runtime) adaptability of components and thus allows a direct connection to real hardware.

Debugging and recording are managed in the last part of the metamodel. *DebuggingSets* are specified, which consist of *RecordingSetups* and *Recordings*. A recording setup executes in the IDE or is translated and configures an embedded platform. In general, a recording can include triggers and timers or can be handled manually. In addition, a *preTrigger* functionality may allow recording of events before the trigger starts by using ring buffers. The recorded values, stored on model level, refer to according parameters and belong to a certain *TimeSlot*. Through the connections in the model the debugging can be completely specified using abstract components and their parameters. In addition, if it is executed on a target platform it can directly take the available abilities into account. Using coding information the recordings can be shown on model level later.

5. MODEL AND EMBEDDED TARGET

In this section we describe the connection and usage between model level and embedded target according to the steps in the development process (see Figure 3).

5.1 Design Time Connection

During design phase the individual components and platforms are stored in the *Database* and combined and configured in the *User-System* model to form the actual system. After design and configuration, code templates of components and platforms are used for generating the source code. The code of the components and thus its behavior adapts to the chosen implementation and the parameter values set. The platform templates are used to form the network between the components and to install fixed components (e.g. for runtime connection or debugging). In our IDE the code templates are written with the Xpand-Framework and adapt by including or excluding code sections and setting values of variables or signals. In the next step the generated code is integrated on the embedded target by a platform specific IDE.

5.2 Runtime Communication

For runtime communication the interface and communication coordinator on the target is either added during the code generation process (see previous subsection) or already present in the system (e.g. debugging interface). The communication takes place using bidirectional messages. These messages are in a predefined format with constant and individual parts including for example the ID of the component as well as the coding of the parameters and their values. The messages are the transport layer of the connection between model and hardware at runtime.

Messages sent from the IDE to the embedded target are generated and sent if the user changes control parameters in the model. On hardware, these messages are directly interpreted and executed by the coordinator. For monitoring, we consider three different ways, which might also be combined depending on the abilities of the platform. The first possibility is to send a message if an event occurs and handle message generation on the embedded target. The advan-

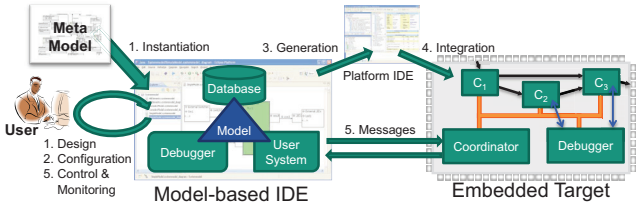

Figure 3: Steps in the development using models for embedded platforms.

tage is that the number of messages directly correlates with the number of events. However, if too many events occur simultaneously in comparison to the communication bandwidth, information is lost in a usually indeterministic way. In the second option, messages are sent on a regular basis, regardless of occurring events. This is deterministic and information can be compressed. However, the reaction time is lower and communication bandwidth is also a limiting factor. The first two methods are independent of the model, as also the IDE, running on a PC, is generally faster than the embedded system. In contrast, the third way polls information from model level: messages are sent from the IDE and the embedded system answers with the respective response. Monitored information of interest can thus be selected by the user in the model during runtime. The disadvantage is the complexity and latency as two messages need to be sent. Furthermore, usually on model level no knowledge on the occurrence of events is present, resulting in a possible loss of data due to a too low update rate.

5.3 Synchronization of Multiple Platforms

One further issue within runtime control and monitoring is synchronization on model level between multiple distributed platforms. This is important if these are not treated individually and interactions between the platforms are examined. In general, synchronization needs to be considered if the message time between IDE and the components on different platforms is highly diverging and if this time becomes greater than the time between two events. If synchronization is not considered the model can become inconsistent, as it does not show the actual status of the system. We consider two different methods to deal with this problem if it occurs. In the first method the frequency of the messages gets limited. Therefore, the update rate for the values gets higher than the communication time for the messages. However, this limits bandwidth and parallelism of events. A synchronization clock and a time stamp in every message can be used as a further possibility [4]. Therefore, every message gets independent and the arrival time is unimportant, if there is no direct runtime representation. The disadvantage is the additional data and that the clocks of the platforms need to be synchronized.

5.4 Control, Monitoring and Recording

To control the embedded system during runtime only the values of the *control* parameters can be changed on model level. Neither can components be added nor can connections be changed. According to changes of the user and coding of parameters and their values in the implementation of the component, corresponding messages with commands are generated and sent to the embedded target.

For monitoring and recording it has to be distinguished between three different possibilities, which we consider in our concept. These options are not exclusive and can be active at the same time. Communication in all methods is performed during runtime as discussed in Section 5.2. Synchronization (Section 5.3) needs to be taken into account if considering a distributed system. In the first option, for direct monitoring, messages from the embedded target are sent during runtime and get directly interpreted and displayed in the user-system model. This is used to get a runtime representation of the status of the embedded target in the model.

The second method records the data in the IDE, i.e. on model level. In large systems, the user can easily concentrate on important aspects by specifying triggers and playback the execution step by step. The main advantage is that for recording and triggering all components and their parameters can be arbitrarily combined on model level to form complex scenarios across multiple platforms. However, limitations of synchronization and bandwidth for the communication apply, because only direct monitored parameters can be recorded in the IDE. Therefore, typically real-time recording is not possible. Using the first two methods only an appropriate interface needs to present on the embedded target, no additional platform hardware is necessary.

For on platform recording, the third method, the available possibilities depend on the abilities of the debugger on the platform. There is usually no inter-platform communication to control the recording and therefore only components on the same platform can be combined. Moreover, trigger, storage and available bandwidth are platform depended and can have limitations. The setup of recording and data transfer is in non-real-time, due to the communication delay. However, as recording is executed exclusively on the platform, it results in possible real-time recording and triggering. Platform recording is specified and configured on model level. This is easily possible as the functionality and handling as well as the connection between model and implementation is known through the other model parts.

All (recorded) status data are similarly interpreted and displayed abstract on the user-system model. The interpretation is performed according to the defined communication messages and monitoring parameters. *NumberParameters* are interpreted and displayed in their respective numerical format and *ListParameters* use the display value and coding value for interpretation. This results in user-optimized abstract access to the system instead of, e.g., cryptic error codes. In addition, we integrate an adaptable graphical representation of information, e.g., the color of a graphical component changes with the value of its parameters. In the recordings the user can navigate step by step forward and backward or jump on events directly.

6. PRACTICAL EXPERIENCES

To test and evaluate our concept we implemented a model-based integrated development environment (IDE) along with examples on a wireless sensor network (WSN) and a reconfigurable hardware platform.

The IDE is based on the metamodel described in Section 4 and can interact with hardware interfaces for runtime con-

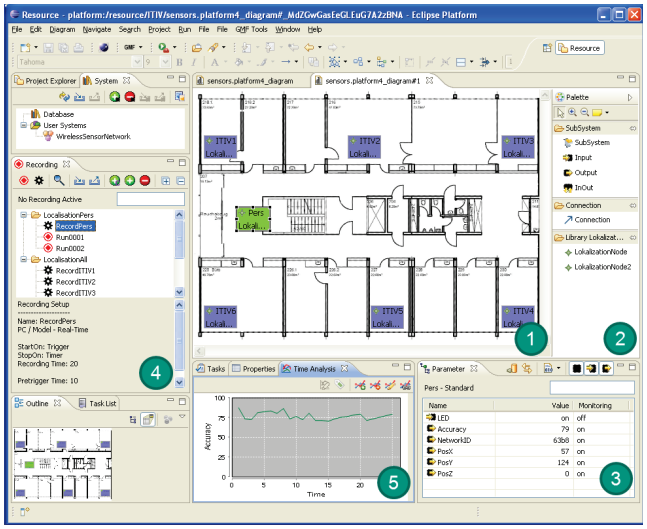

Figure 4: Model-based IDE visualizing a wireless sensor network.

nection to embedded platforms. The IDE is developed on the Eclipse Platform, using the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF). Figure 4 shows the IDE including visualization data of the wireless sensor network example.

In the middle is the graphical modeling area (1), which shows the actual user-system with components and their interfaces and connections. In this example the components are shown in front of a configurable background picture, which represents the environment. The attached tool palette on the right (2) shows the tools for integrating subsystems and connections as well as the components stored in the database. The other views are used to handle the parameters, implementations and platforms (3), connected to the embedded target platforms, manage debugging setups and recordings (4) as well as analyze recorded data (5).

The first example is a distributed WSN, originally intended for the use in indoor ad-hoc localization scenarios [12]. The nodes form a wireless multi-hop mesh network and are programmed in C-language. The IDE accesses the network through a central data sink. We do not manipulate the actual system structure and use the present multi-hop wireless network for communication. In the model the components represent the sensor nodes, which have parameters for configuration, control and show their actual status. For design the parameters configure the attributes of the nodes. In the example, the localization algorithm is controlled and the position of each node monitored during runtime. Therefore, the nodes in the WSN are configured to periodically send their position to the central node connected to the IDE. For the WSN synchronization needs to be considered, because the components run on multiple distributed platforms. Therefore, other parameters are only polled on interest to keep additional traffic as low as possible. For platform debugging we integrate a fixed recording possibility, which stores data on a local SD-card. A message starts the recording and every platform records independently. In practice this results in a small delay (a few ms), which can, however, be neglected for the localization scenario because of the slow location changes of tracked objects. During runtime, the user-system model represents the WSN in the IDE. The position of a node is used for the position of the graphical component in the model in front of an environment picture to represent the location. This enables the model to be a demonstrative representation of the system with direct accessibility for adaptations of the algorithm.

The second example is a reconfigurable hardware platform including a Xilinx Virtex-II Pro Field Programmable Gate Array (FPGA). The components are individual hardware modules implemented in VHDL running on a single FPGA. The example is based on previously published work and has been extended to the new concept [13]. In the template of the platform, besides the component network we integrated an additional small microprocessor and communication network to manage runtime control and monitoring. This raises the complexity, but has the advantage that the component network remains independent. In addition, we integrated our real-time on-chip hardware debugger [14]. During design, the user can assemble, configure and connect different available components in the user-system model. In addition, as the on-chip debugger is integrated in the code generation process, the user needs to set the parameters of interest at design time. During runtime a synchronization problem does not exist as all components are located on a single platform. In practice, there is only a delay of typically less than a half second for messages in both directions, because of the multithreaded IDE and the communication interface. The directly monitored parameters can be selected on model level during runtime. This selection is transmitted once to the microprocessor, which sends a message on every applicable event. This got the advantage of runtime flexibility controlled from the model, but rises the complexity and is also limited by the speed of the microprocessor and the integrated network. For on-chip real-time debugging, recording and triggering are set at runtime, but limited up to a fixed complexity and to the preselected parameters of interest. Thereby, the on-chip debugging is controlled with respect to the components in the model, which represents the system and later show the recorded data.

In both systems, to enable runtime control, monitoring and recording, we instrument the components. This additional effort has to be taken, to get the components as white box modules to control and monitor their internal parameters.

7. CONCLUSIONS AND OUTLOOK

In this paper we presented a concept for design, runtime adjustment and monitoring of embedded systems using component models. We developed a seamless model-based development, especially considering runtime control and monitoring aspects and abilities of embedded hardware platforms. This extends the reuse, scalability and maintainability by abstraction from low-level adjustments and error identification using abstract component models, which comprehensively represents the system at all times. Furthermore, using model level for configuration, control and monitoring eases the usage of different systems from the user's perspective as the user does not necessarily need to know every detail about handling variables or signals on every individual platform.

We introduced a metamodel, which allows the description of components with interfaces, parameters and limitations in combination with execution platforms and their abilities. A platform independent adapted component model is used for design as well as runtime control and monitoring of the actual user-system. Besides the adjustment, also the debugging setup is included on model level. The concept also considers platform specific debugging facilities and interacts with these during runtime to enable, e.g., real-time platform debugging. The monitored data are annotated and integrated into the (graphical) component model. To show the applicability two diverging practical scenarios are chosen. The example WSN application shows the possibility of the integration of a present distributed system with our approach. The second example with reconfigurable hardware showed a range of possibilities in interacting with a highly configurable platform. The limitations of the concept are in conjunction with the design and implementation of the individual components/platforms and the additional necessary information to connect models and hardware. This is especially the case for the considered small embedded platforms not able to run an operating system or middleware.

In the future we want to apply and evaluate our concept to large embedded systems and also include heterogeneous systems with different embedded platforms. Furthermore, we want to integrate more platform functionalities (e.g. in terms of profiling) and communication analysis (e.g. bus load) which can be controlled and monitored from model level. We will also further investigate on the model and integrate composite components and ease the handling and definitions of the components and platforms in the IDE.

8. REFERENCES

[1] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In *Real Time and Embedded Technology and Applications Symposium, RTAS, 11th IEEE*, pages 190–199, Mar 2005.

[2] I. Crnkovic. Component-based Software Engineering for Embedded Systems. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 712–713, USA, 2005. ACM.

[3] W. Damm, A. Votintseva, E. Metzner, and B. Josko. Boosting Re-use of Embedded Automotive Applications Through Rich Components. Proceedings, FIT - Foundations of Interface Technologies, 2005.

[4] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, Dec. 2002.

[5] F. Fleurey, B. Morin, A. Solberg, and O. Barais. MDE to Manage Communications with and between Resource-Constrained Systems. In *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 349–363. Springer Berlin / Heidelberg, 2011.

[6] D. Hammer and M. Chaudron. Component-Based Software Engineering for Resource-Constraint Systems: What are the Needs? In *Object-Oriented Real-Time Dependable Systems, 2001. Proceedings. Sixth International Workshop on*, pages 91–94, 2001.

[7] J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[8] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava. Comparison of Component Frameworks for Real-time Embedded Systems. In L. Grunske, R. Reussner, and F. Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin / Heidelberg, 2010.

[9] A. Koudri, A. Cuccuru, S. Gerard, and F. Terrier. Designing Heterogeneous Component Based Systems: Evaluation of MARTE Standard and Enhancement Proposal. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 243–257, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] M. Lettner, M. Tschernuth, and R. Mayrhofer. A Critical Review of Applied MDA for Embedded Devices: Identification of Problem Classes and Discussing Porting Efforts in Practice. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 228–242, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core Specification, 2004.

[12] J. Schmid, M. Voelker, T. Gädeke, P. Weber, W. Stork, and K. D. Müller-Glaser. An Approach to Infrastructure-Independent Person Localization with an IEEE 802.15.4 WSN. In *Indoor Positioning and Indoor Navigation (IPIN), 2010 International Conference on*, pages 1–9. IEEE, 2010.

[13] T. Schwalb and K. D. Müller-Glaser. Extension of Component-Based Models for Control and Monitoring of Embedded Systems at Runtime. In *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on*, pages 142–148, May 2011.

[14] T. Schwalb and K. D. Müller-Glaser. Model-Based Configuration and Control for Real-Time Debugging on Reconfigurable Hardware. In *System, Software, SoC and Silicon Debug Conference (S4D) 2011, Munich, Germany*, Oct. 2011.

[15] H. Song, G. Huang, F. Chauvel, and Y. Sun. Applying MDE Tools at Runtime: Experiments upon Runtime Models. In *5th International MODELS Workshop on Models@run.time*, Oslo, Norway, 2010.

[16] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An Aspect-Based Composition Tool for Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 58–69, May 2003.

[17] G. Waignier, S. Prawee, A.-F. Le Meur, and L. Duchien. A Framework for Bridging the Gap Between Design and Runtime Debugging of Component-Based Applications. In *3rd International Workshop on Models@runtime*, Toulouse France, 2008.

Model-based Development of Embedded Systems' User Interfaces

Jelena Barth
Albert-Ludwigs-Universität
Freiburg, Germany
barth@informatik.uni-freiburg.de

Bernd Westphal
Albert-Ludwigs-Universität
Freiburg, Germany
westphal@informatik.uni-freiburg.de

Stephan Arlt
Albert-Ludwigs-Universität
Freiburg, Germany
arlt@informatik.uni-freiburg.de

ABSTRACT

We consider the class of embedded systems user interfaces (ES-UI). They differ from classical graphical user interfaces because they use only a limited but possibly multi-modal number of inputs and offer numerous different user interface modes. We propose the domain specific language ESUIL in order to improve the quality of ES-UI software.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE), Evolutionary prototyping, Structured Programming, User Interfaces*; H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces—*interfaces – Graphical user interfaces, Interaction styles, Input devices and strategies, Prototyping, Voice I/O*; J.7 [**Computers in Other Systems**]: Command and Control, Consumer Products

General Terms

Design,Documentation,Languages,Reliability

Keywords

Embedded systems, user interfaces, model-based design, domain specific languages.

1. INTRODUCTION

There are embedded systems which interact not only with a physical plant via sensors and actuators, but also with human users. Examples range from process control systems to the huge domain of consumer electronics where the quality of the user interface is certainly an aspect of the product's success. For instance, contemporary top-of-the-market fridges provide a graphical display where, among others, the temperature of different compartments is monitored and controlled, the operation mode of an ice-maker is chosen, and the user is reminded of the need to change the water filter.

In case of the fridge, all user input is read from a small number of buttons on the front panel as there is no standard keyboard and no mouse. The buttons are assigned different actions depending on the current user interface mode, i.e., the same button, that switches the ice-maker off when the user interface mode is to control the ice-maker, is used to increment the target temperature of the main compartment when the user interface mode is to control the main compartment's temperature.

Technically, the user interface controller is separated from the plant controller. That is, the fridge comprises two embedded controllers: one which is responsible for controlling the plant, e.g., to reach the desired cooling temperature, and one which provides the user interface. The two controllers communicate with each other via a bus interface in both directions. The user interface controller requests the current temperature and sends changed temperature values, the plant controller notifies the user interface of events such as critically high temperatures. In addition, there are many user interface actions which never reach the plant controller, such as changing the current language setting or display brightness, or the usage of integrated egg-timers. This is a classical separation of concerns which makes both controllers easier and allows for an independent development of both.

For us, an *embedded system's user interfaces* (ES-UI) is characterised as follows:

- the ES-UI provides a finite number of *user interface modes* which is indicated to the user by a (possibly graphical or textual) output interface, (such as monitoring temperature, changing target temperature, or changing ice-maker mode),
- in each user interface mode, the ES-UI accepts only a finite number of inputs, that is, there is in particular no standard keyboard, and
- the user inputs are given via a physical input interface which provides only finitely many different events (such as the buttons on the front panel of the fridge),
- the ES-UI interacts with a separate *plant controller* by requesting and sending data, (such as the current and new target temperature in the fridge), and
- the plant controller can notify the ES-UI asynchronously about a finite set of events (such as critically high temperature in the fridge).

Embedded systems are not limited to graphical or textual representations of information. On the one hand, the user interface may simply consist of a single light emitting diode (LED) on the casing and one button. The LED may blink slowly to indicate execution of a certain task, and quickly to indicate that the task failed. Pressing the button shortly may start one task, pressing the button longly another. On the other hand, given the decreasing cost of touchscreens or cameras and microphones and the increasing power of embedded controllers, the in- and output interfaces of embedded system may as well comprise gestures, voice, and finger moves.

Thus, the realisation of an ES-UI can be complex software. ES-UI are nowadays often developed manually [5]. Detecting implementation errors such as unintentional non-determinism, lacking mappings of buttons to internal events, unintentional mappings of buttons in user interface modes where they should not be enabled, etc. by tests is hard as the number of test cases grows exponentially with the number of user interface modes and inputs [6]. Similarly, misunderstandings in requirements are often only identified if at least a mock-up version of the ES-UI is available to the customer.

We propose a model-based approach to ES-UI development. We introduce a domain specific language (DSL) which is tailored for the domain of ES-UI. It provides means to declare the ES-UI interface with the user and the plant controller (the system). The central element of our DSL is the *screen* which corresponds to a user interface mode. Transitions between screens are triggered by system events or user-interface events. To this end, each screen provides a mapping from available buttons to user-interface events. Our DSL promises to improve the quality of ES-UI software in the following ways. Firstly, an ES-UI model can be simulated for validation, that is, instead of using a mock-up for the validation of ES-UI requirements, scenarios can be demonstrated to the customer at hand of the model. Secondly, the ES-UI model may serve as *the* requirements specification communicated between customer and developer. While a simulation for validation can principally also be obtained using *generic*, domain-inspecific modelling languages such as UML, we expect our DSL presented in a visual concrete syntax to be by far more accessible to ES-UI developers, which are today often classical electical engineers without specific education in UML, and also to technically skilled customers. The final software is then considered correct if and only if it has the same behaviour as the model. On the model, quality aspects such as absence of non-determinism, whether there are sufficiently many buttons to map to user-interface events, whether mappings are complete, or whether there are ineffective buttons in any user interface mode can effectively be checked on the ES-UI model as well as more involved functional requirements, for instance, that from each screen, it is possible to reach the main screen again by pressing finitely many buttons. Thirdly, correct ES-UI software in the above sense can automatically be generated from the model. This effectively avoids mistakes which stem solely from the manual approach to implementation. And finally, we imagine our DSL to serve in the documentation of the final product.

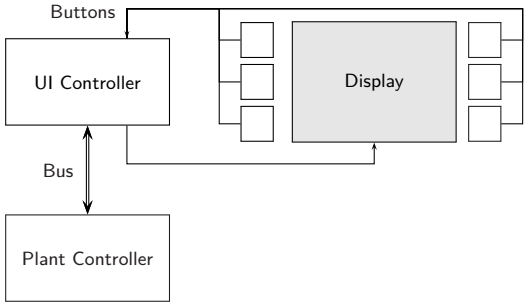

Figure 1: Fridge system architecture.

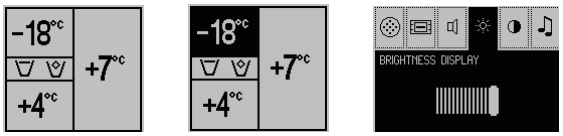

Figure 2: Example screens of user interface modes.

2. EMBEDDED SYSTEMS USER INTERFACES

In order to understand the needs of embedded systems user interfaces, we analysed the user interface of a contemporary top-of-the-market fridge, including the corresponding requirements documents and the software. The company which developed the user interface software for the fridge manufactures provided the documents in a cooperation project. The fridge system is particularly suited for the analysis of embedded systems user interfaces because user interface controller and plant controller are explicitly separated and communicate via a bus. Therefore it is clearly identifiable which behaviour is specific to the user interface. For instance, changing the user interface language should clearly not be a concern of the plant controller.

The user interface consists of a monochrome 254x128 dot matrix LCD display and six buttons, three to the left and three to the right of the display (cf. Figure 1). The buttons are operated as so-called *soft keys*, that is, they have a different effect depending on the user interface mode. Parts at the sides of the LCD are reserved to indicate the effect of each button by an icon, or no icon at all if the button is not effective. The user interface offers different *modes* consisting of a screen layout and a particular mapping of buttons to functions. By default, the display only shows the current target temperature for the three compartments (cf. Figure 2). In this mode, one of the buttons is mapped such that it switches to a mode where the target temperature can be set (cf. second screen in Figure 2). Then and only then, two of the buttons are mapped such that they increment or decrement the target temperature. The remaining user interface modes comprise, for instance, a mode where the display brightness can be set.

The final requirements document for the user interface is given in form of a word processor document. Behaviour is described in the form of informal use cases and scenarios, which are rather paintings than precise specifications. Scenarios consist of sequences of images showing screens as

the ones in Figure 2. Each image comes with an indication which button the user presses in this scenario, the subsequent screen shows the desired effect. For instance, there is a scenario starting with the leftmost screen in Figure 2 and an indication that the topmost button on the right is pressed. The subsequent screen is then the one shown in the middle of Figure 2, that is, in the default user interface mode, it should be possible to switch to the mode where the target temperature is set — we call this a *logical user interface event* — using the topmost button on the right. In addition, there are natural language descriptions of the scenario. Many of the images look almost identical, except for small changes. These changes do not influence the presentation of the display much, i.e. the layout and the available functionality often stay the same to a large amount.

As in many software development projects, the final requirements document is the result of multiple iterations where the developer built a mental, *constructive* [3] model of the behaviour from the given *reflective* description and validated this mental model with the customer. Validation results were integrated into new versions of the word processor document. This form of requirements specification as employed in the considered project was found to be inefficient and error prone by the participating engineers. Inefficient because there were small conceptual changes which caused changes in large parts of the document and error prone because scenarios were found to be conflicting or to be under-specifications of the actual requirements. Visual aspects of the user interface were largely excluded from this requirements elicitation process. The final code is developed using the *mental*, constructive model of the user interface behaviour which is established in the requirements validation phase. The constructive model is hardly documented; if at all, then in form of code fragments.

In the source code which realises the user interface of the fridge, one can identify a clear division into responsibilities. There is a module which deals with the bus communication, a module which realises the logical behaviour of the user interface, i.e., the mapping of buttons to functions and switching between user interface modes, and there is a module which deals with the graphical design of the user interface, that is, which renders, e.g., the temperature figures, icons, and text on the display. The implementation was done manually. The resulting code is of high quality (well-structured, well documented) yet the developers see the risk of mistakes. For example, the mapping of buttons to functions is realised via arrays of function pointers. When using copy-and-paste as usual, there is a certain risk that buttons remain mapped although they shouldn't be or to assign the wrong icon (depicting the user interface event) to a button. A mistake which is difficult to detect in tests; testers may not consider to try buttons which are unmapped according to the display, but erroneously mapped according to the function pointer array.

Following [11], a user interface model can be structured into three different models. The *System Model* "represents the physical system". The *GUI Logical Model* "is a model of the GUI in its behavioural aspects. We don't have its actual visual representation here, but only its abstraction in terms of interaction — what objects make it up, what are their structural parameters, what they accept and return, and how they interface with the objects of the system in terms of commands and values." The *GUI Visual Model* "models the visual aspects of the GUI and the associated semantics." Interactions take place between System and GUI Logical Model and between GUI Logical and GUI Visual Model.

We find a corresponding structure in the software of the fridge user interface. The plant controller is the *system layer*, the module of the user interface controller software which realises the logical behaviour of the user interface is the *UI logical layer*, and the module which renders icons and text on the display is the *UI visual layer*. For the development of the fridge user interface, the focus was on the UI logical layer. The system layer was already existing with a well-defined interface. A more detailed model of the behaviour of the plant controller was not necessary. The negotiation of the requirements interface was (strictly speaking) conducted independently from the graphical representation.

3. ESUIL

To improve the quality of user interface software with respect to validity ("are we building the user interface which is desired by the customer?") and correctness, we propose to offer means to the developer which allow her to make the *mental*, constructive model of the user interface behaviour, the UI logical model, which is established during the requirements validation phase explicit.

These means should on the one hand allow for easier and more precise communication with the customer, for instance by offering live simulation to demonstrate scenarios, and on the other hand offer code generation facilities in order to avoid errors which stem from the manual implementation such as unintentionally mapped buttons. To this end, we assume that the interface via which user interface and plant controller communicate is given. Furthermore, we propose to consider graphical design aspects of the ES-UI only in a very abstract form.

Our new domain specific modelling language introduced here as the desired means is supposed to be employed in a model-based user interface development process (MBUID) [10]. We propose to view an ES-UI on the UI logical layer as a finite set of user interface modes or *screens*. A screen need not be associated with any information about objects or widgets on it. Transitions between screens are triggered by *logical events* which we distinguish from *physical events* to support the development of ES-UIs with narrow input interface. Examples for a single physical event are pressing the topmost button on the right on the fridge, or pressing the two lowermost buttons at the same point in time. Physical events may be mapped to logical events. For example, the physical event "topmost button on the right pressed" is mapped to the logical event "switch user interface mode" in the default screen of the fridge. Supporting logical events as well as physical events is necessary because the mapping between both determines the behaviour of the final user interface. Furthermore, realising the mapping between both is a potential cause of errors. An explicit representation of the mapping allows us to verify the model already in the requirements validation phase, e.g., by automatically checking whether, for each screen, there are enough physical events for all logical

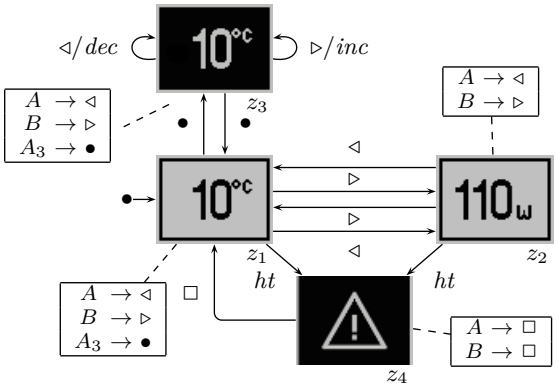

Figure 3: ESUIL model of the simplified fridge.

events. Furthermore, considering physical and logical events separately provides a degree of freedom regarding modality. The ES-UI model for the fridge with six buttons could easily be re-used for a fridge with a touchscreen by just changing the mappings.

In the following, we introduce the abstract syntax and semantics of the domain specific modelling ESUIL.

To characterise the input-/output interface of the UI-controller (cf. Figure 1), we firstly assume that the set of notifications by the plant controller to the UI controller and, e.g., data requests from the UI controller to the plant controller are given as system-out and system-in events, respectively. Secondly, we assume that the set of possible user inputs, the so-called physical events, is given. And finally, we assume that a set of logical or UI-events is available. Formally, we group these sets in a signature defined as follows.

Definition 1. An *ES-UI signature* is a quadruple

$$\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$$

where $\mathcal{U}$ is a set of *UI-events*, $\mathcal{S}_{out}$ is a set of *system-out events* disjoint to $\mathcal{U}$, $\mathcal{S}_{in}$ is a set of *system-in events*, and $\mathcal{P}$ is a set of *physical events*. ♢

Note that we require system-out and logical events to be disjoint because they will be used as triggers of edges and we want distinguish whether edges are triggered by user inputs or system outputs.

The user interface of the fridge is much too large to serve as a complete running example. So we consider a simplified fridge with only two buttons A and B and a display. In addition, we assume that we can detect that button A was pressed for at least 3 seconds before being released. The simplified fridge basically offers two monitoring user interface modes, one showing the current target temperature of the single compartment and one showing the current average energy consumption. From the mode showing the current target temperature, a mode can be reached where the buttons increment and decrement the target temperature. In addition, the fridge can display a warning if the plant controller detects a critically high temperature in the compartment.

To model the simplified fridge, we choose $\mathcal{P} = \{A, B, A_3\}$ corresponding to the inputs we distinguish for the buttons, $\mathcal{S}_{out} = \{ht\}$ for the high temperature warning from the plant controller, $\mathcal{S}_{in} = \{inc, dec\}$ to increment and decrement the target temperature, and we choose the user interface events $\mathcal{U} = \{\square, \bullet, \triangleleft, \triangleright\}$.

Definition 2. An *ES-UI model* is a structure

$$\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$$

where

- $\mathcal{Z}$ is a finite set of *screens*, $z_{ini} \in \mathcal{Z}$ is the *initial screen*,
- $\mathcal{I}$ is an ES-UI signature,
- $\mathcal{E} \subseteq \mathcal{Z} \times (\mathcal{U} \cup \mathcal{S}_{out}) \times (\mathcal{S}_{in} \dot\cup \{\cdot\}) \times \mathcal{Z}$ is a set of directed edges. An element $(z, \varepsilon, \alpha, z') \in \mathcal{E}$ describes an edge from screen z to screen z' labelled with the *trigger* $\varepsilon \in \mathcal{U} \cup \mathcal{S}_{out}$ and the *action* $\alpha \in \mathcal{S}_{in} \dot\cup \{\cdot\}$, where "$\cdot$" is the dedicated *nil-action* which is not supposed to appear in $\mathcal{S}_{in}$, and
- $\Psi : \mathcal{Z} \to (\mathcal{P} \nrightarrow \mathcal{U})$ is a function that assigns to each screen a physical-to-UI-events mapping. A *physical-to-UI-events mapping* assigns each screen $z \in \mathcal{Z}$ a partial function $\Psi(z) : \mathcal{P} \nrightarrow \mathcal{U}$ (also denoted by ψ_z for short) which maps physical events to UI-events. ♢

To model the simplified fridge, we use four screens z_1 (monitor target temperature), z_2 (monitor energy), z_3 (change target temperature), and z_4 (high temperature warning). z_1 is the initial screen. The edges are given by Figure 3. For instance, the right self-loop on z_3 represents $(z_3, \triangleright, inc, z_3)$. It models the desired behaviour that the UI-event $\triangleright$ is enabled in screen z_3 and has the effect that the system event *inc* is sent to the plant controller. Physical-to-UI-events-mappings are given by the tables in Figure 3. For instance, in screen z_1, all three physical events are mapped to different UI-events. Pressing A for more than three seconds in z_1 enables the mode z_3. In the warning screen z_4, both buttons A and B are mapped to $\square$ which confirms the warning. Pressing A for more than three seconds has no effect in z_4.

Definition 3. Let $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ be an ES-UI model with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$.

A finite or infinite sequence $\pi = z_0 \xrightarrow[\alpha_0]{\varepsilon_0} z_1 \xrightarrow[\alpha_1]{\varepsilon_1} z_2 \xrightarrow[\alpha_2]{\varepsilon_2} \dots$ with $z_i \in \mathcal{Z}$, $\varepsilon_i \in \mathcal{U} \cup \mathcal{S}_{out}$, and $\alpha_i \in \mathcal{S}_{in} \dot\cup \{\cdot\}$, is called *UI-computation of $\mathcal{M}$ starting in z_0* if and only if, for all $i \in \mathbb{N}_0$, there is an edge $(z_i, \alpha_i, \varepsilon_i, z_{i+1})$ in $\mathcal{E}$.

A UI-computation of $\mathcal{M}$ starting in the initial screen z_{ini} is called *UI-computation of $\mathcal{M}$*. ♢

The sequence

$$\pi = z_1 \xrightarrow[\cdot]{\triangleright} z_2 \xrightarrow[\cdot]{\triangleright} z_1 \xrightarrow[\cdot]{\bullet} z_3 \xrightarrow[dec]{\triangleleft} z_3 \xrightarrow[dec]{\triangleleft} z_3 \xrightarrow[\cdot]{\bullet} z_1 \xrightarrow[\cdot]{ht} z_4 \xrightarrow[\cdot]{\square} z_1$$

is a UI-computation of the simplified fridge model. It corresponds to firstly checking the energy consumption, then

again the temperature, decrementing the target temperature by two units, monitoring the temperature again, and finally confirming a high-temperature warning. Note that high-temperature warnings get lost in screen z_3.

Definition 4. Let $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ be an ES-UI model with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$.

A finite or infinite sequence $\iota = z_0 \xrightarrow[\alpha_0]{p_0} z_1 \xrightarrow[\alpha_1]{p_1} z_2 \xrightarrow[\alpha_2]{p_2} \dots$ with $z_i \in \mathcal{Z}$, $p_i \in \mathcal{P}$, and $\alpha_i \in \mathcal{S}_{in} \dot\cup \{\cdot\}$, is called *(user) interaction of $\mathcal{M}$ starting in* z_0 if and only if there is a UI-computation

$$z_0 \xrightarrow[\alpha_0]{\varepsilon_0} z_1 \xrightarrow[\alpha_1]{\varepsilon_1} z_2 \xrightarrow[\alpha_2]{\varepsilon_2} \dots$$

of $\mathcal{M}$ starting in z_0 where for all $i \in \mathbb{N}_0$, either $\varepsilon_i \in \mathcal{S}_{out}$, or ψ_z is defined for physical event p_i, i.e. $p_i \in \mathrm{dom}(\psi_{z_i})$, and $\varepsilon_i = \psi_{z_i}(p_i)$.

An interaction of $\mathcal{M}$ starting in the initial screen z_{ini} is called *(user) interaction of $\mathcal{M}$*. ♢

A user interaction corresponding to π is

$$\iota = z_1 \xrightarrow[\cdot]{B} z_2 \xrightarrow[\cdot]{B} z_1 \xrightarrow[\cdot]{A_3} z_3 \xrightarrow[dec]{A} z_3 \xrightarrow[dec]{A} z_3 \xrightarrow[\cdot]{A_3} z_1 \xrightarrow[\cdot]{ht} z_4 \xrightarrow[\cdot]{A} z_1.$$

Definition 5. Let $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ be an ES-UI model with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$.

A screen $z \in \mathcal{Z}$ is called *UI-reachable* in $\mathcal{M}$ if and only if there exists a UI-computation

$$\pi = z_0 \xrightarrow[\alpha_0]{\varepsilon_0} z_1 \xrightarrow[\alpha_1]{\varepsilon_1} z_2 \xrightarrow[\alpha_2]{\varepsilon_2} \dots \xrightarrow[\alpha_n]{\varepsilon_n} z_n$$

of $\mathcal{M}$ such that $z_n = z$.

z is called *interaction-reachable* in $\mathcal{M}$ if and only if there exists an interaction

$$\iota = z_0 \xrightarrow[\alpha_0]{p_0} z_1 \xrightarrow[\alpha_1]{p_1} z_2 \xrightarrow[\alpha_2]{p_2} \dots \xrightarrow[\alpha_n]{p_n} z_n$$

of $\mathcal{M}$ such that $z_n = z$. ♢

Screen z_3 is UI- and interaction-reachable in the simplified fridge model. Screen z_4 is also UI- and interaction-reachable, but each user interaction which reaches z_4 comprises an earlier system-out event.

4. QUALITY CRITERIA

In the following, we use our formal definition of ESUIL to formalise four quality criteria for ES-UI designs. Designs which do not satisfy these quality criteria are commonly considered erroneous in ES-UI development. All properties are efficiently decidable for ESUIL models as checking whether an ESUIL models satisfies these quality criteria amounts to screens and outgoing edges once.

Definition 6. An ES-UI $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$ is called *deterministic* if and only if there does not exist a screen with different outgoing edges having the same trigger, i.e. if

$$\forall\, (z_1, \varepsilon_1, \alpha_1, z_1'), (z_2, \varepsilon_2, \alpha_2, z_2') \in \mathcal{E} \bullet$$
$$z_1 = z_2 \wedge \varepsilon_1 = \varepsilon_2 \implies \alpha_1 = \alpha_2 \wedge z_1' = z_2'.$$

Otherwise, $\mathcal{M}$ is called *non-deterministic*. ♢

Definition 7. Let $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ be an ES-UI model with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$. The signature $\mathcal{I}$ is called *sufficient* wrt. $\mathcal{M}$ if for each screen, the number of physical events in $\mathcal{P}$ is large enough to be mapped to all UI events occurring at outgoing edges, i.e. if

$$\forall\, z \in \mathcal{Z} \bullet |\{\varepsilon \in \mathcal{U} \mid \exists\, \alpha, z' \bullet (z, \varepsilon, \alpha, z') \in \mathcal{E}\}| \leq |\mathcal{P}|.$$

Otherwise, $\mathcal{I}$ is called *too narrow* wrt. $\mathcal{M}$. ♢

Definition 8. An ES-UI $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$ is called *fully mapped* if and only if there does not exist an edge in $\mathcal{M}$ which is triggered by a UI-event that is not in the range of the physical-to-UI-events mapping of the source screen, i.e. if

$$\forall\, (z, \varepsilon, \alpha, z') \in \mathcal{E}\ \exists\, p \in \mathcal{P} \bullet \psi_z(p) = \varepsilon.$$

Otherwise, $\mathcal{M}$ is called *partially mapped*. ♢

Definition 9. An ES-UI $\mathcal{M} = (\mathcal{Z}, z_{ini}, \mathcal{I}, \mathcal{E}, \Psi)$ with ES-UI signature $\mathcal{I} = (\mathcal{U}, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathcal{P})$ is said to have *effective physical events* if and only if for each screen, if there is a physical event mapped to a UI-event ε, then there is a corresponding outgoing edge with trigger ε, i.e. if

$$\forall\, z \in \mathcal{Z}, p \in \mathcal{P}, \varepsilon \in \mathcal{U} \bullet$$
$$\psi_z(p) = \varepsilon \implies \exists\, \alpha \in \mathcal{S}_{in} \dot\cup \{\cdot\}, z' \in \mathcal{Z} \bullet (z, \varepsilon, \alpha, z') \in \mathcal{E}.$$

Otherwise, $\mathcal{M}$ is said to have *ineffective physical events*. ♢

As UI- and interaction-reachability are decidable for ES-UIL models, which reduce to finite state machines, we can employ LTL or CTL to state more general requirements. Examples are additional generic quality criteria such as the absence of UI- or interaction-unreachable screens, or, in case of the simplified fridges, the design specific requirement that the initial screen is interaction-reachable from all screens in the model.

5. CODE GENERATION

From an ESUIL model, a skeleton for the user interface software can be generated which refers to an abstract interface (1) to the system, (2) to the physical events, and (3) to the UI visual layer. For the first case, the code simply assumes a method or function called like the system event. In case of the simplified fridge, these would be the functions *inc* and *dec*. These functions need an implementation which communicates with the plant controller. Similarly, we assume an interface to the plant controller which emits system-out events such as *ht* in the simplified fridge.

For the second case, we assume an interface which reads the buttons and generates pre-filtered physical events. For

example in the simplified fridge, we don't plan to generate the code which measures whether button A has been pressed for more than three seconds. Yet such a code generation can easily be integrated into ESUIL by assuming a description language over buttons which allows us to denote, e.g., button combinations.

For the third case, we require a redraw method or function per screen. Each time, a screen is entered via a transition, the framework calls the redraw method. The implementation of the redraw method is supposed to realise the visual UI design decisions.

6. CASE STUDY

We have used an extension of ESUIL with hierarchical screens to model the user interface of the top-of-the-market fridge. The model covers top-level user interface modes, different modes for changing the target temperature of the different compartments, and the user interface modes for fridge and user interface settings.

7. RELATED WORK

There is a huge body of work on model-based development of user interfaces which targets user interfaces which we term as *desktop* user interfaces (ranging from early works like [4] to more recent works like [9]). The difference to embedded systems user interfaces is that user interaction takes place via *virtual* controls that are part of the graphical output medium. Most prominently widgets that can be clicked on using a mouse, or by shortcuts defined on a standard keyboard. Consequently, there is no need of mapping (physical) buttons to user-interface events, the virtual button *is* the user-interface event. Our approach in contrast considers *physical* controls. We support the development of user interfaces for embedded systems which are controlled by, e.g., physical buttons on the panel of the device that the computer system is embedded into.

For instance [7], supports the development of interactive applications for complex process control. Process control software is supposed to monitor and adjust exactly the values of the controlled process. They require a sophisticated structural model of the controlled process. The declaration of system events can be seen as a very simple instance of such a model. Furthermore, our approach is not limited to process control. Strictly speaking there need not even be a controlled process, then the sets of system-in and -out events are empty.

Closer to our approach is Dygimes [2]. They assume a task model in form of a ConcurTaskTree (CTT) [8], which we don't assume. The DSL we propose here could be used as an intermediate language in the Dygimes approach. The BATIC^{3}S approach [1, 11, 12] also assumes a CTT task model. Our approach adopts the idea of distinguishing three levels of abstraction for an UI model (cf. Section 2).

The authors of [5] give an overview over the current situation and the most important achievements in the last 30 years of model-based user interface development (MBUID) from their point of view. The overview is based on the Cameleon Reference Framework (CRF), which "serves as a reference for classifying UIs that support multiple targets, or multiple contexts of use on the basis of a model-based approach". In terms of CRF, our approach provides an interface structure for all these layers of abstraction to different extents in the following sense. Firstly, our model is supposed to fully cover "Tasks & Concepts". From "Abstract UI" we support the logical part, the placement of objects on the screen is of no concern for us but this information can be supplemented, e.g., by attaching it to our screens. In the "Concrete UI" (CUI) phase, our concrete physical events are determined. ESUIL contributes to "Final UI" (FUI) in form of code generation,

8. CONCLUSION AND FURTHER WORK

Based on a thorough analysis of the needs of ES-UI developers, we propose the new domain specific language ESUIL as a means to improve the quality of ES-UI software. Our formal definition of the abstract syntax and the semantics of the new domain specific language ESUIL allows us to precisely define common errors in ES-UI development. This is a first step towards avoiding these errors in the future.

Future work first of all comprises a full elaboration of ESUIL. For ESUIL to be practically useful, we need to work out an appealing concrete syntax, that is, ESUIL models should be presented as shown in Figure 3. The fridge case study shows that ESUIL needs to be extended by hierarchical states similar to statecharts. Furthermore, code generation needs to be elaborated to be able to apply ESUIL to more case studies.

9. REFERENCES

[1] V. Amaral et al. BATIC^{3}S project document collection. Technical Report 1, Univ. Genève, 2006.

[2] K. Coninx et al. Dygimes. In *Mobile HCI*, pages 256–270. Springer, 2003.

[3] D. Harel. Some thoughts on statecharts, 13 years later. In O. Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 226–231. Springer, 1997.

[4] P. Johnson et al. ADEPT. In *CHI*, page 56. ACM, 1993.

[5] G. Meixner et al. Past, present, and future of MBUID. *i-com*, 10(3):2–11, 2011.

[6] A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.

[7] F. Moussa, C. Kolski, and M. Riahi. A model based approach to semi-automated user interface generation for process control interactive applications. *Int. with Co.*, 12(3):245–279, 2000.

[8] F. Paternò et al. ConcurTaskTrees. In *INTERACT*, pages 362–369. Chapman & Hall, 1997.

[9] A. Puerta et al. UI Fin. In *IUI*, pages 345–354. ACM, 2009.

[10] A. R. Puerta. A model-based interface development environment. *IEEE Softw.*, 14:40–47, 1997.

[11] M. Risoldi and V. Amaral. Towards a formal, model-based framework for control systems interaction prototyping. In *RISE*, pages 144–159. Springer, 2007.

[12] M. Risoldi et al. A language and a methodology for prototyping user interfaces for control systems. In D. Lalanne and J. Kohlas, editors, *HMI*, pages 221–248. Springer, 2009.

Formal Execution Semantics for Asynchronous Constructs of AADL

Jiale Zhou, Andreas Johnsen, Kristina Lundqvist
School of Innovation Design and Technology
Mälardalen University, Västerås, Sweden
{zhou.jiale, andreas.johnsen, kristina.lundqvist}@mdh.se

ABSTRACT

The Architecture Analysis and Design Language (AADL) has been widely accepted to support the development process of Distributed Real-time and Embedded (DRE) systems and ease the tension of analyzing the systems' non-functional properties. The AADL standard prescribes the dispatching and scheduling semantics for the thread components in the system using natural language. The lack of formal semantics limits the possibility to perform formal verification of AADL specifications. The main contribution of this paper is a mapping from a substantial asynchronous subset of AADL into the TASM language, allowing us to perform resource consumption and schedulability analysis of AADL models. A small case study is presented as a validation of the usefulness of this work.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

General Terms

Reliability, Verification

Keywords

AADL, TASM, verification, formal methods, formal semantics

1. INTRODUCTION

Distributed Real-time and Embedded (DRE) systems deployed for instance on avionics and aerospace platforms is one of the most safety-critical categories of systems. Usually, DRE systems consist of many local subsystems. Compared with more traditional all-in-one systems, distributed systems tend to have a larger number of non-deterministic aspects. Therefore, designing distributed systems demands more control during the development phases and the use of rigorous methodologies. Moreover, ensuring that the produced system conforms to all stringent functional and non-functional requirements is a very complex and time consuming task. For instance, one common headache with DRE systems is how to, with a high degree of trust, analyze the impact of event triggered aperiodic/sporadic threads to its local subsystem and verify the functional and non-functional requirements of the local system under this circumstance. The model- and component-based development approaches have emerged as attractive options for the development of DRE systems. The Architecture Analysis and Design Language (AADL) [16] has been widely accepted to support the development process of DRE systems and ease the tension of analyzing the systems' non-functional properties. However, the lack of formal semantics limits the possibility to perform formal verification of AADL specifications. Although efforts have been made towards specifying formal semantics of AADL [1, 3, 7–11, 17, 18] there are still some open questions left. For instance, asynchronous interactions, i.e., aperiodic and sporadic threads, are to our knowledge not covered. Within this context, we are motivated to consider an asynchronous subset of AADL in our work of providing a formal semantics of AADL.

We have chosen Timed Abstract State Machine (TASM) [14] as the language to define the formal semantics. TASM is a novel specification language, which has been shown the potential to express formal semantics of AADL [15]. Especially, two distinctive features make TASM stand out. Firstly, TASM supports the specification of both functional and non-functional behavior. The non-functional properties that can be expressed include timing behavior and resource consumption. Secondly, the TASM toolset provides procedures for analysis of completeness, consistency, execution time and resource consumption. Analysis of time-related properties is provided through a translation into timed automata – the input language for the UPPAAL model-checker [2].

The main contribution of this paper is a translation of a chosen subset of AADL into TASM, allowing us to perform resource consumption and schedulability analysis. and schedulability analysis of AADL models. A small case study is presented to show how AADL models can benefit from this work. The rest of the paper is organized using the following structure: Brief overviews of AADL and TASM are presented in Section 2 and Section 3, respectively. Section 4 describes the formal semantics for the chosen subset of AADL. Section 5 presents the corresponding transformation rules. Section 6 shows a case study applying the translation and performing the resource consumption and schedulability

ACES-MB 2012 Innsbruck, Austria

analysis. Some concluding remarks can be found in Section 7.

2. A BRIEF OVERVIEW OF AADL

AADL was released and published as a Society of Automotive Engineers (SAE) Standard AS5506 in 2004 [16]. It is a textual and graphical language, which describes the architecture of component-based systems as an assembly of software components mapped onto components representing the execution platform.

Data, subprograms, threads, threads group and *processes* collectively represent application software components. *Processor, memory, bus* and *device* collectively represent the execution platform. Execution platform components support the scheduling and execution of threads, the storage of data and code, and the communication behavior between processes. *Systems* are called compositional components. They allow software and execution platform components to be organized into hierarchical structures with well-defined features. AADL offers an execution model that addresses most of the runtime-needs of real-time systems: (1) a set of execution model properties can be attached to each AADL declaration; (2) the semantics of the execution model is also described, namely, the execution semantics of AADL. However, most of it is defined using a natural or semi-formal language. The absence of a precise mathematical semantics makes any pretense of achieving formal verification meaningless [11].

2.1 The Chosen Subset of AADL

The chosen subset includes AADL thread and processor components. AADL thread component is the only component with execution semantics in AADL. In the chosen subset of AADL, an AADL thread can be periodic, aperiodic, or sporadic. Periodic thread dispatches are solely determined by the time interval specified through the *Period* property value. An aperiodic or sporadic thread dispatch is triggered non-deterministically. But for sporadic threads, a minimum interval time between successive dispatches has to be specified through the *Period* property value. The property *Priority* specifies the execution order when more than one threads are ready to execute. The range property *Compute_Execution_Time* defines the Best Case Execution Time (BCET) and Worst Case Execution Time (WCET). For brevity, we only consider WCET in the paper. AADL processor component is an abstraction of the runtime environment and execution platform, where a scheduler is implicitly included. In this paper, we use the terms scheduler and processor interchangeably. The scheduler plays the role in coordinating all thread executions on one processor as well as concurrent access to shared resources. Various scheduling protocols can be specified according to the *Scheduling_Protocol* property value. In this paper, we consider a preemptive fixed-priority scheduler.

Definition 1. An AADL-specification A is a pair <Pr, T> where:

- Pr is a processor, which is a triple <Ident, T_Bind, Sch_Protocol>:
  - Ident denotes the identifier of the processor, which must be unique in the range of the specification.
  - T_Bind denotes a set of threads bound to the processor, where $T\_Bind \subseteq T$.
  - Sch_Protocol denotes the $Scheduling\_Protocol$ property. We assume that the value of the property is "preemptive fixed-priority".
- T is a set of thread components. Let t range over T. A thread t_i is a pair <Id_i, $\mathrm{Sch\_Prop}_i$>:
  - Id_i denotes the identifier of the thread t_i which must be unique in the range of the specification.
  - $\mathrm{Sch\_Prop}_i$ denotes a set of scheduling properties. More specifically, $\mathrm{Sch\_Prop}_i = <Dispatch\_Protocol_i, Compute\_Execution\_Time_i, Compute\_Deadline_i, Priority_i, Period_i>$ of the form $Property ::= Identifier => Value$. We assume that the value of the $Dispatch\_Protocol$ can possibly be "aperiodic", "sporadic", and "periodic".

3. A BRIEF OVERVIEW OF TASM

TASM [14] was born at MIT, USA, and now the toolset is being extended at Mälardalen University, Sweden. TASM is a formal language for the specification of embedded real-time systems. The TASM language extends the Abstract State Machine (ASM) [5] to enable the expression of timing and resource consumption.

Definition 2. A TASM specification is a pair <E, ASM> where:

- E is the environment, which is a triple <EV, TU, ER>:
  - EV denotes Environment Variables, the global variables that affect and are updated by machine execution,
  - TU denotes the Type Universe, a set of types that includes real numbers, Integer, Boolean, and user-defined types,
  - ER denotes Environment Resources, a set of named resources. More specifically, ER={(r_n, r_s) | r_n is the resource name, and r_s is the resource size}. Examples of resources include memory, power, and bus bandwidth.
- ASM is the abstract state machine, which is a 4-tuple < MV, CV, IV, R>:
  - MV denotes Monitored Variables, the set of environment variables that affects the machine execution,
  - CV denotes Controlled Variables, the set of environment variables that the machine updates,
  - IV denotes Internal Variables, the set of local variables and they are visible merely inside the machine,
  - R denotes a set of Rules, R={<n,t,RR,r> | n is the rule name; t specifies the duration of a rule execution, which can be a single value or a range value [t_{min}, t_{max}] or the keyword *next*, the *next* construct essentially states that time should elapse until one of the other rules is enabled. Especially, the lack of a time annotation is assumed to mean t = 0; RR is the resource consumption

during the rule execution. Similarly, the omission of a resource consumption annotation is assumed to mean zero resource consumption; r is a rule of the form "if *guard* then *action*", where guard is an expression depending on the monitored or internal variables, and action is a set of updates of the controlled or internal variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled.}.

As an extension of ASM, TASM describes system behaviors as the computing steps of an abstract machine with time and resource annotations. The basic execution semantics of a TASM machine is described as follows: In one step, it reads the monitored variables, selects a rule of which guard is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it non-deterministically selects one to execute. In TASM, time progresses in a fixed constant step called a clock tick which is the minimum time quota. As a specification language, TASM supports the concepts of hierarchical composition, parallelism, communication and synchronization. Hierarchical composition is achieved by means of auxiliary machines - function machines and sub machines. Parallelism is naturally supported, since TASM main machines are executed in parallel. Communication and synchronization between machines can be achieved by communication channels [13] whose semantics is similar to the concept of rendezvous in the Ada programming language.

4. FORMAL SEMANTICS FOR THE SUBSET OF AADL

In this section, the formal semantics for the chosen subset of AADL is presented in TASM. Firstly, we present the formal semantics for the AADL thread component, which can be regarded as two subcomponents - *Dispatcher* and *Thread*. For each sub-component, the formal semantics is described in terms of the sub-component's states and a corresponding TASM main machine. Secondly, we present the formal semantics for the scheduler in the form of its possible states and a TASM main machine with several Auxiliary Machines (AM).

4.1 AADL Thread

In AADL, periodic, aperiodic, and sporadic threads have the same life cycle [4] but different dispatch protocols. Therefore, we regard the thread component as *Dispatcher* and *Thread*.

4.1.1 Dispatcher

As its name implies, *Dispatcher* represents the behavior of a dispatch protocol which can be periodic, aperiodic, or sporadic according to the *Dispatch_Protocol* property value.

Dispatcher can have two possible states - *dispatch* (initial state) and *wait*. The *dispatch* state denotes that a dispatch of the thread is triggered immediately (if the thread is periodic) or after a non-deterministic time duration (if the thread is aperiodic or sporadic). The *wait* state denotes that *Dispatcher* is waiting for the elapse of a specified period to send the next request. In the *Dispatcher* model, a state variable, *disState*, is used to denote the current state of the dispatcher. Its initial value is *dispatch*.

Dispatcher main machine consists of five rules, as shown in Listing 1. *Rule Dispatch* changes the dispatcher state from *dispatch* to *wait* and sends a dispatch request through a global variable *disFlag* to *Thread*. We use a variable *timer* to trace the elapsed time between dispatches. *Rule NonDeterministic* does nothing, but costs 1 clock tick. When the modeled thread is aperiodic or sporadic, *Rule NonDeterministic* and *Rule Dispatch* are always enabled or disabled simultaneously. As a reminder, in TASM, if more than one rules in the same machine are enabled simultaneously, solely one of them will be non-deterministically selected to execute. We introduce this inconsistency purposely to simulate the non-deterministic scenario of dispatching aperiodic and sporadic threads. When the modeled thread is periodic, *Rule NonDeterministic* is always disabled. Both *Rule Waiting* and *Rule WaitComplete* cost 1 clock tick. They are used to simulate the *wait* state of the dispatcher. For a periodic or sporadic thread's dispatcher, *Rule Waiting* is enabled when the period of the corresponding thread does not elapse. On the contrary, *Rule WaitComplete* is enabled when the period has elapsed and updates its state to *dispatch*. For an aperiodic thread's dispatcher, the specified period is zero, so both rules are always disabled. The last rule, *Rule Else* is used to keep the machine alive, in case no other rules are enabled.

4.1.2 Thread

Thread is responsible for modeling the execution semantics of AADL threads once they are dispatched. Within the AADL context, the complete AADL thread execution model incorporates complex functional and non-functional behaviors. For brevity and simplicity, our model solely focuses on basic functional behaviors - thread dispatching, thread scheduling and execution, but ignores mode transition, remote subprogram, data communication and error recovery. However, these behaviors are subjects for future work.

The possible undergoing thread states can be simplified into *awaiting_dispatch* (initial state), *compute*, and *complete*. The *awaiting_dispatch* state denotes that a thread is awaiting a dispatch request. The *compute* state denotes a thread is currently computing. The *complete* state denotes a thread completes its computation and returns to the *awaiting_dispatch* state. More detailed, the *compute* state can be further refined into two states - *ready* and *running*. The *ready* state denotes that a thread is awaiting the allocation of necessary resources for performing the upcoming execution, such as memory or CPU-time. The *running* state denotes that a thread is currently occupying the CPU and being executed. In the *Thread* model, two hierarchical state variables are used - *thdState* and *thdCmpState* which respectively describe the current thread state and the current refined *compute* state. The initial value of *thdState* is *awaiting_dispatch*. The initial value of *thdCmpState* is *none* which merely denotes the thread is not being executed.

The execution semantics of a thread is expressed as a main machine with six rules, as is shown in Listing 2. *Rule WaitDispatch* is enabled when *Thread* is in the *awaiting_dispatch* state and a dispatch request is received. It changes the state of *Thread* from *awaiting_dispatch* to *compute*, and updates the *thdCmpState* variable to be the *ready* state. *Rule ComputeReady* blocks the *Thread* machine until a signal through *runThd* channel is received from *Scheduler* that also updates the *Thread* machine to the *running* state. A thread within

the *compute* state may be subjected to preemption, where its time and resource consumption must be stalled. TASM does not allow a rule execution to be interrupted by any other rule. In order to model the behavior of preemption, *Rule ComputeRunning* and *ComputeComplete* are defined. Both *Rule ComputeRunning* and *ComputeComplete* cost 1 clock tick. When the thread is in the *running* state, *Rule ComputeRunning* is enabled repeatedly when the amount of elapsed clock ticks is less than WCET-1. Rule *ComputeComplete* is enabled when the amount of elapsed clock ticks is equal to WCET-1, and then changes thread state to the *complete* state. *Rule Complete* is used to complete the current dispatch of the thread. Currently this rule solely changes the thread state back to the *awaiting_dispatch* state, but will be used to implement additional actions of data communication and shared resources in future work. *Rule WaitNextDispatch* is used to model the idle time between dispatch requests.

4.2 Scheduler

A scheduler grants *Thread* to execute on the processor based on the specified priority scheme. It ensures that only one thread is being executed on a particular processor. If no thread is in the *ready* state, the scheduler is idle until at least one thread enters the *ready* state. A thread will remain in the *running* state until it completes execution of the dispatch or until a thread with higher priority entering the *ready* state preempts it. The execution semantics varies according to its scheduling protocol. In this section, we present the execution semantics of a preemptive fixed-priority scheduler.

A scheduler has three possible states - *wait_thread* (initial state), *sche_thread*, and *run_thread*. The *wait_thread* state denotes that the scheduler is awaiting until a thread enters the *ready* state. The *sche_thread* state denotes that the scheduler selects one thread with the highest priority from the set of threads in the *ready* state. The *run_thread* state denotes the scheduler grants the selected thread to execute. In the *Scheduler* model, a state variable designated *schState* traces the state of *Scheduler*.

The *Scheduler* main machine makes use of five auxiliary machines, both sub machines and function machines, as is shown in Table 1. Due to limited space, we do not present them in detail in this paper. The execution semantics of a scheduler is modeled as a main machine with five rules, as shown in Listing 3. *Rule WaitThread* is enabled when at least one new thread enters the *ready* state or if there is any thread left in the *ready* state when the processor is released. Then it updates the scheduler to the *sche_thread* state. *Rule ScheThread* is enabled when the scheduler is in the *sche_thread* state. It selects the thread with the highest priority from the set of threads in the *ready* state. *Rule PreemptThread* is enabled if the selected thread has a higher priority than the currently running thread. And the sub machine *RUNNEXTTHD()* is called to execute. On the contrary, *Rule RunThread* is enabled if the running thread has a higher priority. This rule changes the *Scheduler* machine to the *wait_thread* state. *Rule Idle* is used to keep the machine alive when no other rules are enabled.

5. TRANSFORMATION TO TASM

Based on the definition of AADL and TASM presented in Section 2 and Section 3 and the formal semantics presented in Section 4, we define two transformation rules for AADL thread and processor component. For the sake of the limitation of pages, we solely show the main part of the rules in Listing 1, 2, 3. The transformation rules are expressed in the form of the LET-IN construction:

- *entity* =
  LET $element_1 = body_1$
  AND $element_2 = body_2$ ...
  IN <$element_1$, $element_2$, ...>
  END *entity*

where the *elements* between the angle brackets conform to the formal definition of *entity*.

AM	Type	Description
isNewReadyExist	Function	return true if a new thread becomes *ready*, else false
isCPUFree	Function	return true if the CPU is currently free, else false
isReadyExist	Function	return true if there is any *ready* thread, else false
scheduleThreads	Function	return the highest priority thread among the threads in the *ready* state
Preemptted	Function	return true if the running thread is preemptted
RUNNEXTTHD	Sub	suspend the running thread and execute the next thread

Table 1: The Auxiliary Machines (AM) Used by the Scheduler Main Machine

```
t_i =
 LET TASM_Dispatcher(i) =
  LET E_disp =
   LET TU_disp = DisState := {dispatch, wait};
       ThdType := {periodic, aperiodic, sporadic};
   IN <EV_disp, TU_disp, ER_disp>
  AND ASM_disp =
   LET R_disp =
    Dispatch{ if disState_i=dispatch then disState_i
        := wait; disFlag_i:= dispatched; timer:= 0;}
    NonDeterministic{ if disState_i = dispatch and
        disProtocol_i != periodic then skip;}
    Waiting{ if disState_i = wait and timer<(
        thdPeriod_i-1) then timer := timer +1;}
    WaitComplete{ if disState_i=wait and timer=(
        thdPeriod_i-1) then timer := timer +1;
        disState_i := dispatch;}
    Else{ t:=next; else then skip;}
   IN <MV_disp,CV_disp,IV_disp,R_disp>
  IN <E_disp, ASM_disp>
```

Listing 1: Transformation Rule of AADL Thread

6. CASE STUDY

In order to illustrate how AADL models can benefit from our formal semantics, we present a case study of the verification of an adapted version of the follower spacecraft guidance system (FSGS) example presented in [6].

6.1 Follower Spacecraft Guidance System

```
AND TASM_Thread(i) =
 LET E_thread =
  LET TU_thread = DisPatchFlag := {none, WithOutRes,
       WithRes}; ThreadState:= {awaiting_dispatch
      , compute, complete}; ThreadComputeState :=
       {none, ready, running};
  AND ER_thread = power := [POWER_SIZE]; memory
       := [MEM_SIZE];
  IN <EV_thread, TU_thread, ER_thread>
 AND ASM_thread =
  LET R_thread =
   WaitDispatch{ if thdState_i=awaiting_dispatch
       and disFlag_i=dispatched then thdState_i:=
       compute; thdCmpState_i:= ready; disFlag_i:=
       notdispatched; cmpTime:= 0;}
   ComputeReady{ if thdState_i=compute and
       thdCmpState_i=ready then runThd_i?;}
   ComputeRunning{ t:= 1; power:=POWER_
       CONSUMPTION; memory:=MEM_
       CONSUMPTION; if thdState_i=compute and
        thdCmpState_i=running and cmpTime<thdWCET_i
       -1 then cmpTime:=cmpTime+1;}
   ComputeComplete{ t:= 1; power:= POWER_
       CONSUMPTION; memory:=MEM_
       CONSUMPTION; if thdState_i=compute and
        thdCmpState_i=running and cmpTime=thdWCET_i
       -1 then thdState_i:=complete; cmpTime:=
       cmpTime+1; thdCmpState_i:=complete;}
   Complete{ if thdState_i=complete then thdState_i
       :=awaiting_dispatch; thdCmpState_i:=none;}
   WaitNextDispatch{ t:= next; else then skip;}
  IN <MV_thread,CV_thread,IV_thread,R_thread>
 IN <E_thread, ASM_thread>
IN TASM_Dispatcher(i) || TASM_Thread(i)
END t_i
```

Listing 2: Transformation Rule of AADL Thread (cont'd from Listing 1)

```
Pr =
 LET TASM_Processor =
  LET E_processor =
   LET TU_processor = ScheState := {wait_thread,
       sche_thread, run_thread};
   IN <EV_processor, TU_processor, ER_processor>
  AND ASM_processor =
   LET R_processor =
    WaitThread{ if scheState = wait_thread and (
       isNewReadyExist() or (isCPUFree() and
       isReadyExist())) then scheState :=
       sche_thread;}
    ScheThread{ if scheState = sche_thread then
       scheState := run_thread; nextRunThread :=
       scheduleThreads();}
    PreemptThread{ if scheState = run_thread and
       Preemptted() then scheState := wait_thread;
        RUNNEXTTHD();}
    RunThread{ if scheState = run_thread and !
       Preemptted() then scheState := wait_thread;
        nextRunThread := none;}
    Idle{ t:= next; else then skip;}
   IN <MV_processor,CV_processor,IV_processor,R_processor>
  IN <E_processor, ASM_processor>
 IN TASM_Processor
END Pr
```

Listing 3: Transformation Rule of Scheduler

Thread	Period	WCET	Priority	Power	Memory
Receiver	100	10	high	20	20
Reader	100	20	middle	30	10
Watcher	100	30	low	50	30

Table 2: Parameters of Threads in FSGS

FSGS consists of three threads. A sporadic thread (*Receiver*) receives position data which is sent periodically from the leader spacecraft, updates its own position data, and sends the position data to the *Reader* thread. A *Reader* thread reads periodically the position value from the *Receiver* thread and stores it in a protected object. A *Watcher* thread "watches and reports" the object to the earth observation station. This model is a typical sub system of a distributed system, with a sporadic thread to exchange data and a set of periodic threads devoted to process data. We assume that all the threads need resources - power and memory, which is shown in Table 2.

6.2 TASM model

The *Scheduler* machine schedules the execution order of threads based on fixed-priority scheduling protocol. All the threads are hard real-time threads, that is, a missed deadline is regarded as a system failure. The model of the periodic threads *Reader* and *Watcher* are respectively expressed by two main machines (*Dispatcher* and *Thread*) with the parameters listed above. Although a sporadic thread can theoretically be triggered at any time after a minimum period, we assume a maximum period within which the *Receiver* thread will be triggered at least once. The maximum period can be the hyper-period of the periodic threads or any other reasonable value. This assumption is reasonable, because as long as the follower spacecraft does not deviate from the leader spacecraft, the FSGS will receive the position data within a maximum period.

6.3 Verification and Validation

6.3.1 Resource Consumption

We use the TASM toolset to analyze resource consumption of the FSGS system. As depicted in Figure 1, the graph shows the aggregate resource consumption in the first period of the FSGS system for each resource - power (upper) and memory (lower), versus the horizontal time axis. Three distinctive high levels represent the resource consumption of the corresponding threads. Because the FSGS system does not contain any parallelism consumption of resources, the minimum and maximum amounts of resources consumed will correspond to the minimum and maximum amounts contained in an individual thread.

6.3.2 Timing Properties

The TASM machines can easily be translated into Timed Automata through the transformation rules defined in [12]. The transformation enables the use of the UPPAAL model checker to verify the schedulability of the FSGS system. In addition, deadlock freedom is an essential property that should be satisfied, which also is a prerequisite for schedulability analysis. Table 3 shows the queries of the properties and the corresponding results.

7. CONCLUSION AND FUTURE WORK

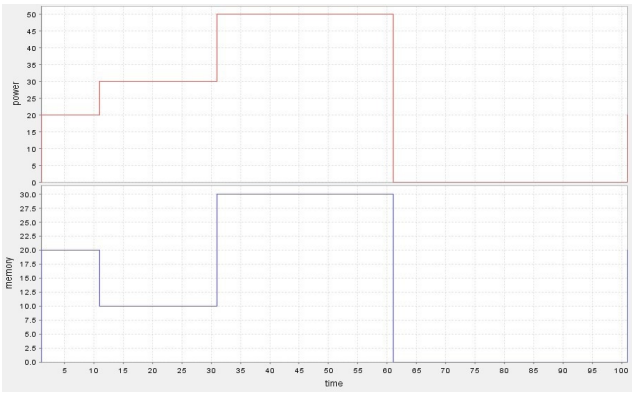

Figure 1: Resource Consumption (resources on the Y-axes and time on the X-axes)

Property	Query	Result
Deadlock Freedom	*A[] not deadlock*	Satisfied
Schedulability	*A[] not Reader.MissDeadline or Watcher.MissDeadline or Receiver.MissDeadlin*	Satisfied

Table 3: Deadlock Freedom and Schedulability Analysis for FSGS

We present an approach to provide formal resource consumption and schedulability analysis for AADL models of a local subsystem of a DRE system. The approach is to translate the execution semantics of AADL components into rule machines in the TASM language. Periodic, aperiodic and sporadic threads and a preemptive fixed-priority scheduler are covered. We purposely introduce inconsistent rules into the translated TASM machine in order to model the non-deterministic aspects. A small case study is conducted to show how to perform resource consumption and schedulability analysis. Resource consumption analysis is enabled by using the TASM toolset. Schedulability analysis of the translated TASM model is carried out by mapping the TASM model into timed automata.

Future work on this approach will cover a larger subset of AADL components, such as, additional components, the Behavioral Annex, mode change, data communication, the Error Annex, etc. Additional scheduling protocols will be incorporated for analysis and evaluation.

Acknowledgement

This work was partially supported by the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre (MRTC)/Mälardalen University.

8. REFERENCES

[1] T. Abdoul, J. Champeau, P. Dhaussy, P. Y. Pillain, and J.-C. Roger. AADL Execution Semantics Transformation for Formal Verification. In *ICECCS '08*, pages 263–268. IEEE Computer Society, 2008.

[2] G. Behrmann, R. David, and K. G. Larsen. A tutorial on UPPAAL. pages 200–236. Springer, 2004.

[3] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Zilio, M. Filali, and F. Vernadat. Formal Verification of AADL Specifications in the Topcased Environment. In *Ada-Europe '09*, pages 207–221, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] J.-P. Bodeveix, R. Cavallero, D. Chemouil, M. Filali, and J.-F. Rolland. A mapping from AADL to Java-RTSJ. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 165–174. ACM, 2007.

[5] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.

[6] M. Y. Chkouri and M. Bozga. Prototyping of Distributed Embedded Systems Using AADL. In *ACES-MB '09*, pages 65–79, October 2009.

[7] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Models in Software Engineering. chapter Translating AADL into BIP - Application to the Verification of Real-Time Systems, pages 5–19. Springer-Verlag, Berlin, Heidelberg, 2009.

[8] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *EMSOFT '07*, pages 134–143, 2007.

[9] A. Johnsen. An Architecture-based Verification Technique for AADL Specifications. Technical Report, Mälardalen University, January 2012.

[10] D. Monteverde, A. Olivero, S. Yovine, and V. Braberman. VTS-based Specification and Verification of Behavioral Properties of AADL Models. In *Model Based Architecting and Construction of Embedded Systems*, 2008.

[11] P. C. Ölveczky, A. Boronat, and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In *FMOODS/FORTE '10*, pages 47–62, 2010.

[12] M. Ouimet. *A formal framework for specification-based embedded real-time system engineering.* MIT, Dept. of Aeronautics and Astronautics, 2008.

[13] M. Ouimet and K. Lundqvist. The TASM Toolset: Specification, Simulation, and Formal Verification of Real-Time Systems. In *Computer Aided Verification*, volume 4590, pages 126–130. Springer Berlin / Heidelberg, 2007.

[14] M. Ouimet and K. Lundqvist. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. In *RTNS '07*, 2007.

[15] L. Pi, Z. Yang, J.-P. Bodeveix, M. Filali, K. Hu, and D. Ma. A Comparative Study of FIACRE and TASM to Define AADL Real Time Concepts. In *ICECCS '09*, pages 347–352, 2009.

[16] SAE. Architecture Analysis & Design Language (AADL). SAE Standards AS5506, November 2004.

[17] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *IPDPS '06*, 2006.

[18] Z. Yang, K. Hu, D. Ma, and L. Pi. Towards a formal semantics for the AADL behavior annex. In *DATE '09*, pages 1166–1171, 2009.

Automatic synthesis from UML/MARTE models using channel semantics

Pablo Peñil
University of Cantabria
ETSIIT, Los Castros s.n.
39005 Santander, Spain
+34 942 200878
pablop@teisa.unican.es

Héctor Posadas
University of Cantabria
ETSIIT, Los Castros s.n.
39005 Santander, Spain
+34 942 200878
posadash@teisa.unican.es

Alejandro Nicolás
University of Cantabria
ETSIIT, Los Castros s.n.
39005 Santander, Spain
+34 942 200878
nicolasa@teisa.unican.es

Eugenio Villar
University of Cantabria
ETSIIT, Los Castros s.n.
39005 Santander, Spain
+34 942 201398
villar@teisa.unican.es

ABSTRACT
Model-driven design is very common nowadays. In this context, the UML/MARTE profile is a well-known solution for real-time, embedded system modeling. This profile enables the functional and non-functional details of the system to be modeled together. Regarding non-functional details, the profile allows certain real-time constraints to be imposed when describing the system concurrency, in order to ensure predictability. However, these constraints also limit the modeling flexibility required to evaluate different design alternatives when optimizing system performance. The paper proposes a solution for automatically synthesizing the resulting models, combining new communication semantics with standard UML/MARTE real-time management features. The UML/MARTE approach presented in this paper enables concurrency and synchronization effects to be modeled at communication points, making system exploration and implementation easier.

Categories and Subject Descriptors
C.0 [**Computers system organization**]: General – *HW/SW Interfaces and Systems specification methodology*

General Terms
Performance, Design, Languages.

Keywords
Synthesis, code generation, design exploration, UML modeling.

1. INTRODUCTION
The evolution of fabrication technologies has enabled the development of complex Systems on Chips, containing multiple HW components, such as CPUs, DSPs or GPUs. As a result, electronic devices have evolved, integrating large amounts of functionality into a single product.

The need for development of the corresponding, growing SW, has led designers to increase the abstraction level of the first stages of the design process. Designing at higher levels of abstraction, designers find an effective way to deal with large system complexity, selecting optimal configurations and verifying system constraints early in the design process. However, the more oriented the design steps are to ensuring the fulfillment of system constraints, the more difficult it is to find flexible solutions enabling the exploration of different design alternatives to obtain optimal results.

In this context, model-driven design methodologies are commonly adopted to handle the design of these embedded systems. The latest design methodologies start from high-level UML models combined with algorithmic codes (e.g. C, C++, Matlab, etc.) of the different components of the system [1]. These UML models are commonly developed following different profiles oriented to add additional semantics to the original basic UML components. Among these profiles, MARTE is gaining increasing interest for the development of real-time, embedded systems.

The development of real-time systems becomes increasingly complex as the number of processing elements in the platform grows. The resulting parallelism makes it more difficult to guarantee the fulfillment of all the real-time constraints. To solve these difficulties, the MARTE profile places the concurrency modeling inside the models of the functional components. For this purpose, it proposes different solutions, such as limiting the generation of dynamic threads or controlling the amount of threads that can run in parallel in active components.

However, these limitations can reduce the flexibility, which is required to model and explore the performance effects of defining different concurrency architectures for the system. In order to optimize system performance, the effects of creating different execution flows and deciding different HW resource allocations have to be evaluated. Thus, system-modeling methodologies and implementation flows should be flexible enough to enable evaluations of multiple design decisions with minimal designer effort.

To be able to confront the evaluations of multiple design decisions with minimal designer effort, two main issues have to be solved. First, it is necessary to provide methodologies in which designers can easily describe and modify the system and its concurrency and allocation details, considering all the interactions among its functional components. Additionally, solutions capable of evaluating and implementing the different models in an automatic or semi-automatic way are required.

In this context, the paper proposes, first, an extension of MARTE communication media with additional semantics that enables different concurrency architectures to be modeled in a flexible way. Secondly, the paper shows how the executable files required

to perform the optimization design flow are automatically generated using the information included in the UML model.

In order to present the proposed approach, the paper is structured as follows. First, the state of the art is presented. After that, the UML/MARTE modeling methodology applied is described, focusing on the new elements required for modeling concurrency support in channels, and their combination with other elements previously proposed in MARTE. After that, details about the automatic synthesis process are shown. Then, the application of the proposed solution to a real example is presented and, finally, conclusions are stated.

2. STATE OF THE ART

The interest of the automatic synthesis of code from high-level models has been increasing in recent years. High-level models, both graphical, such as models based on UML, and executable models, such as SystemC-based models, have been proposed for their use at the beginning of the design process.

In the SystemC context, [6] presents a generic framework for HW/SW communication of functional tasks with shared resources. Communication is implemented using a method-based interface implementing a RMI protocol. In a similar way, the Embedded System Environment (ESE) is presented in [5]. The ESE consists of an application model mapped to a given platform enabling a automatic TLM (Transaction-level modeling) of the system. From this TLM, the software and hardware required to produce the cycle-accurate model (CAM) are synthesized.

Additionally, in [7], a method for systematic embedded software generation from SystemC is presented. The SW code (processes and process communication, including HW/SW interfaces) is systematically generated, from SystemC threads. And in [20], the HetSC methodology is presented. HetSC is a system specification methodology for concurrent heterogeneous embedded systems in SystemC, supporting the creation of formal executable specifications of the system.

All these works take advantage of the executable capabilities of SystemC to generate real executable implementations. As a result, the designer's effort is reduced. However, there are two main disadvantages. First, these approaches use specification codes in SystemC as implementation codes, something which is not always the most suitable solution. Furthermore, the use of executable codes as a starting point limits the exploration capabilities of the approach, since the evaluation of any modification in concurrency or allocation details usually requires the code to be modified. In order to overcome this, graphical inputs based on UML models have been proposed.

The application of UML models for embedded system design has gained increasing interest ([1] and [2]). UML models have been demonstrated to be an adequate approach for generating high-level models from which implementation codes can be obtained. Several synthesis processes based on UML models are characterized by the creation of state machine models or variations of them [8]. In [9], a set of transformation rules for synthesizing code from UML activity diagrams is presented. UML Sequence diagrams are used to define control flow patterns.

However, not only system functionality can be obtained form UML models. In fact, several UML-based methodologies are also focused on synthesis of HW/SW communications. In [10] a semi-automatic solution for generation of HW/SW infrastructure from UML models is presented. This solution implements a high-level programming interface (software drivers and hardware adapters) using Remote Method Invocation (RMI) semantics as the framework to unify the communication interfaces for all HW and SW components. All the SW and HW adapters are automatically generated from functional descriptions of the interfaces.

In [11], a method is proposed for synthesizing interfaces for heterogeneous IP (Intellectual property) integration from UML models. The framework supports both interface protocol customization and glue logic generation, thereby maximizing IP integration. Additionally, the framework enables the generation of communication links among the system blocks from UML profiles used to model the system-level communication interfaces.

However, UML usually lacks all the semantics required to adequately model all the characteristics of embedded systems. In order to confront the challenge of covering the complete design flow of real-time embedded systems, the MARTE profile was created [4]. Taking MARTE-based models, the methodology proposed in [19] describes embedded concurrent real-time applications. The model is transformed into an executable platform through the code generation Accord|UML.

In [17] and [18] a MARTE-based methodology for abstract and formal specification of heterogeneous systems, supporting a safe use of concurrency and the automated generation of SystemC models, is introduced.

Taking MARTE-based models as input, several synthesis approaches have also been proposed. Gaspard2 [12] is a design environment for data-intensive applications which enables MARTE description of both, the application and the hardware platform, including MPSoC and regular structures. Through model transformations, Gaspard2 is able to generate an executable TLM SystemC platform at the timed programmers view (PVT) level.

In [9] the complete design flow to move from high-level MARTE models to code generation, for implementation of dynamically reconfigurable SoCs is presented. In this paper, generic control semantics for the specification of adaptive and dynamically reconfigurable SoCs is presented. In [13] a design flow based on high-level languages (SysML, MARTE, SystemC…) enables the generation of the deterministic multi-threaded code for parallel implementations.

However, all the previous solutions are oriented to generating completely fixed models, especially in their concurrent structure. Design space exploration of concurrency and allocation are not considered, requiring large efforts from the designers to find optimal design solutions. Even [24], which supports different resource allocations, does not provide enough capability to explore the optimal system concurrent architecture.

To solve this problem, this paper presents a methodology to help the designers to explore different design alternatives. The approach enables to design the concurrent structure in the UML/MARTE model, according to the communication semantics defined in the communication mechanism. Then, automatic synthesis is performed to obtain the executable files required to obtain performance evaluations of the resulting architectures. This exploration of the concurrent structure of the system enables performance optimizations to be obtained that provide better final implementations.

3. MODELING CONCURRENCY

As is well-known, task parallelization has a critical impact on system performance. Defining the most adequate concurrency architecture and the most appropriate resource allocation are critical issues when optimizing system performance. However, current UML/MARTE facilities do not facilitate this optimization at the high level.

In UML/MARTE models, concurrency is mainly described within the functional components, while channels have no properties to describe the behaviour semantics at application level. For instance, the attributes of the MARTE stereotype <<CommunicationMedia>> enable to describe physical properties of the channel (latency, capacity…), but no architectural properties such as sequential or parallel execution of the requested service, size of communication buffers, etc.. Components can be defined to be active or not, fixing different characteristics, such as the possibility of generating dynamic threads and the maximum number of threads that can run in parallel in order to support the different activities that can be performed in parallel. Additionally, activity or sequential diagrams can be used to describe how the different threads interact among themselves.

Considering a system as a set of functional components that operates requiring services from each other, concurrency can be generated in the system in two main ways: as a result of communication issues, or as a result of the internal execution of a certain service. If functional codes are directly provided by the designer, together with the UML model, the latter case is fixed in the code, so any modification in this kind of concurrency, requires manual modification of this code. Code related to thread creation and, even, the functions' algorithms themselves can require deep modifications. Thus, these optimizations are more focused on implementation steps than on initial system development and will not be considered in this paper.

However, the former case can be fully managed from the UML model, enabling optimization of the system performance at the high level. For real-time purposes, the maximum number of concurrent threads for each component is usually fixed within the models of the system components. However, it is not always easy to find the optimal number of threads for each component.

Execution flows are usually generated in active components as a resource that reacts to incoming requests. As a result, several limitations can appear. First, in complex systems, it is not always easy to know the number of threads that can be required in a certain component at a time. Secondly, it can be difficult to balance the number of threads among the different components, in order to obtain maximum parallelism in the system.

Furthermore, as concurrency is defined within the components, all services provided by a single component operate in the same way. Then, it is no easy to model and evaluate different combinations of communication behaviors between clients and servers.

The proposal presented in this paper is to add behavior semantics to the communication channels, combined with the information described in the functional components. In this way, the properties of the channels indicate when threads and synchronization points must be created, while the models of the components control the maximum number of active threads in order to avoid an explosion in the number of concurrent threads, an explosion that would make the design completely unpredictable.

Using this approach, the different services of each component can operate with different characteristics. As a result, different options of system concurrency architecture can be easily described in the model, enabling later design space exploration. New channel behavior semantics enable the designer to indicate when incoming requests can be forced to wait until service completion, waiting until the service starts or letting the client continue. Additionally, buffers oriented to storing the data to be transferred can be specified in the model, indicating the buffer size.

As a result, different types of common channels can be modeled. For example, connections through FIFOs can be described by letting the client leave the data in the channel buffer and continuing its execution without waiting for the server response.

This approach enables different design alternatives to be explored just requiring minimal modifications in the model. Additionally, the increase of communication semantics can simplify the modeling of designs previously described with different approaches, such as pure functional executable descriptions, Kahn networks, approaches which usually rely on service calls or specific communication channels. Design optimizations such as pipelines can also be easily evaluated in order to increase the concurrency of the code. Finally, the association of communication semantics with communication channels simplifies the automatic synthesis processes, as this association enables the concurrency control to be implemented within the communication wrappers.

The application of this communication semantics can be applied together with other standard real-time constraints included in the functional components of the MARTE profile. The proposal to enable both semantics to be combined will be described in the section dedicated to communication synthesis.

4. PROPOSED UML/MARTE MODELING

The UML/MARTE methodology followed in this paper applies the idea of separation of concerns as a solution to create models that are easy to develop and explore. This separation is achieved by providing the designer with distinct system views. Specifically, system views enable the modeling of the PIM by identifying the application components, the functionality implemented by each of these application components and the different communication mechanisms used to interconnect these application components.

As a result, the UML/MARTE methodology is component-based; the system is an entity that is composed of a set of components which are interconnected through a well-defined interfaces. The component-system composition makes the system to be clearly separable in reusable blocks, improving reusability and modularity. Moreover, this UML/MARTE methodology enables to model the set of memory partitions where the different application components can be allocated. Finally, designer can model the HW/SW platform in order to specify the allocation of the memory partitions into the HW/SW resources.

This UML/MARTE methodology is also software centric since application components are allocable units on the HW/SW platform. Finally, this UML/MARTE methodology applies the modeling foundations that Model-Driven Architecture [16] provides for the development of the HW/SW embedded systems. Specifically, in this paper we focus on the modeling of the Platform Independent Model (PIM). However, the main contributions of this paper are focused on the modeling of the

application components and the communication mechanisms which interconnect these application components.

4.1 Application Components

The application components are modeled by the MARTE stereotype <<RtUnit>>. Some attributes of the <<RtUnit>> stereotype are considered in this methodology. We define that the value of attribute *isDynamic* should be true to specify that the application component creates threads dynamically in order to attend the requests for services provided by the *RtUnit*. However, the attribute *srPoolSize* should be defined by a specific value in order to denote that *RtUnit* has a finite set of threads in order to deal with the request to the services provided by *RtUnit*. According to the previous attribute values, the attribute *srPoolPolicy* should be *infiniteWait* to denote that the *RtUnit* waits infinitely till a thread finishes dealing with a service request in the case that *RtUnit* does not have more threads available. The *isMain* attribute can be considered to denote the main application.

RtUnits uses an UML port specified by the MARTE stereotype <<ClientServerPort>> in order to define the interfaces which communicate with other application components. The interfaces are modeled by the MARTE stereotype <<ClientServerSpecification>>. Each interface includes a set of necessary functions to enable communication among application components. The functions can be *re-entrant*, *protected* and *concurrent* [15]. A modeling constraint is established in the specification of the interface functions: an interface can only have functions of a specific type.

4.2 Communication Components

Communication mechanisms are modeled by the MARTE stereotype <<CommunicationMedia>>. However, the semantics of the *CommunicationMedia* has been enriched by some additional characteristics, specifically, characteristics used to model the communication semantics associated with one application component that uses the communication media to access a service provided by another application component. These additional semantic characteristics are captured by applying the stereotype <<CommSpecification>> defined in the proposed methodology that extends the semantics of the <<CommunicationMedia>> stereotype.

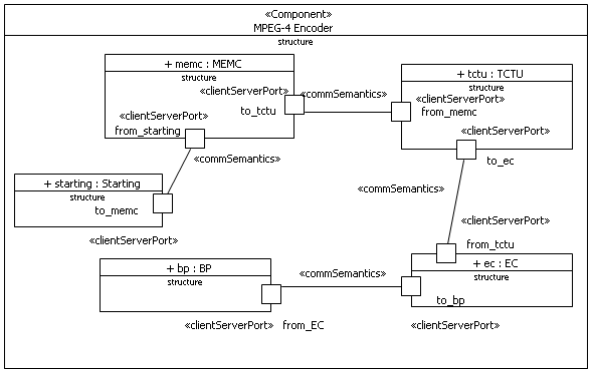

Figure 1. PIM.

The stereotype *CommSpecification* has the attributes *blockingFunctionCalls, blockingFunctionReturn, priority* and *timeout.* The attribute *blockingFunctionCalls* defines whether the client application is blocked until the server application attends the service request. The attribute *blockingFunctionReturn* defines whether the client application is blocked waiting for the response of the service called. The attribute *priority* defines the priority associated with the application client in order to attend its service request by the server application. The attribute *timeOut* defines the maximum time that the client waits for the function response. An additional characteristic can be associated with the *CommunicationMedia*. By applying the MARTE stereotype <<StorageResource>> the CommunicationMedia can store function call requests, defining the size of this storage capacity through the attribute *resMult* (when the *resMult* attribute is not defined, there is understood to be an infinite storage capacity).

Each application component can be a structured component. The internals of an application consist of *RtUnit* instances interconnected by an UML connector. The assembly connectors should be specified by the stereotype <<CommSemantics>> defined by this methodology. In this way, specific communication semantics is associated with a connector (Figure 1). Thus, the communication semantics that specifies the interconnection among application components is defined by the kind of interfaces required/provided by the application components and the value of the attributes of the stereotypes *StorageResource* and *CommSpecification.* The delegation connectors should not be specified by the *CommSemantics* stereotype.

5. SYNTHESIS PROCESS

Once the UML/MARTE model of the system is created, the next step is to implement the system captured through an automatic synthesis. For this purpose, this approach proposes an automatic process capable of generating the communication interfaces, and then the executable files. These files can be used to evaluate the performance of the system. In fact, according to Figure 2, automatic synthesis processes are required for two different goals: enabling the evaluation of the different design alternatives in order to define the optimal design depending on concurrency and resource allocations, and obtaining the final implementation process. That combination of performance evaluation and automatic code generation enables handling system constraints.

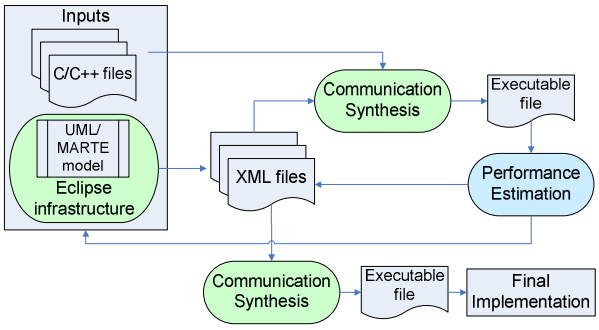

Figure 2. Proposed Synthesis flow.

The definition of a process oriented to exploring different design alternatives requires a flexible development flow. For this purpose, the flow proposed is divided into four steps, according to Figure 2. First, the designer develops the UML/MARTE model, providing the functional C codes of the different components at the same time. From the UML model, several XML files are generated, containing all the information captured in the model. These files enable automatic synthesis of all the communication wrappers required to build the final system. These wrappers, together with the functional C files, are combined to create the executable files that are used to provide the performance

estimations and the final implementation. The executable files are created using compilation scripts that are also generated from the UML model.

Performance estimations of the system can be provided using virtual platform models, such as [21],[22] or running the code on the target platform, to run the executable files obtained. These estimations are used to decide which design solution is the best according to the set of system requirements. In order to take this decision, two alternatives can be used: the designer can manually explore and analyze the results, modifying the UML/MARTE model appropriately, or an automatic exploration tool [23] can be used to evaluate all the possibilities. In the latter case, the exploration is performed by modifying the XML files instead of the graphical model itself. Nevertheless, in order to simplify the proposal, only manual exploration is considered in this paper.

In this paper, we focus on showing how, with a few modifications in the UML/MARTE model, different design alternatives can be specified. Specifically, modifications in the communication semantics of the system channels generate different system implementations. These model modifications involve the synthesis of different communication interfaces in order to implement the system. As a consequence of the designer can establish a complete, easy-to-use system performance exploration procedure, defining the system metrics to be explored.

The synthesis of the communication interfaces requires codes capable of implementing the channel semantics described in the model. Thus communication interfaces must handle data transfers, synchronization and concurrency according to the model.

In order to perform data transfers, synchronization and concurrency, channels must contain three functional sections: a section receiving the client request, saving data and performing adequate synchronization, a second section performing communications between the client side and server side, and a third section, generating concurrency in the server, and executing the server services, when requested. These operations are integrated into the client wrapper and the server wrapper, which are automatically synthesized from the XML files (Figure 3).

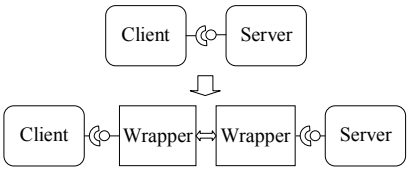

Figure 3. Channel insertion in the system.

In order to maintain the real-time characteristics of the model, the maximum number of threads specified for each *RtUnit* is also considered. To support this, all threads created or resumed in the servers are bounded by a protection mechanism. A semaphore is used to control the accesses performed to each service call, ensuring that no more than a certain number of threads are operating concurrently at a particular moment.

Finally, other controlling mechanisms have also been implemented in order to ensure the correctness of the communications, such as package identifiers and acknowledgement packages.

6. MPEG4 ENCODER EXAMPLE

The UML/MARTE approach proposed has been applied to a demonstrative example. This example is an MPEG-4 encoder based on the MPEG-4 Simple Profile reference code, which has been pruned and cleaned for the intended purposes.

The structure of the MPEG-4 encoder consists of a set of functional blocks. Functional blocks which can be recognized in the code are: the functional block MEMC (MotionEstimation-MotionCompensation), the TCTU (TextureCoding-TextureUpdate), EntropyCoding (EC) and BitstreamPacketizing (BP). He MPEG-4 encoder is modeled as Figure 4 shows. . An additional application block is included in the application model. The RtUnit component "Starting" represents the functionality that triggers the MPEG encoder system. All the application components are interconnected through interfaces accessing these interfaces through communication media.

In order to evaluate the validity of the proposed approach for the exploration of the optimal design, some design configurations have been considered in the MPEG-4 implementation. These alternatives are based on different synchronization and concurrency architectures.

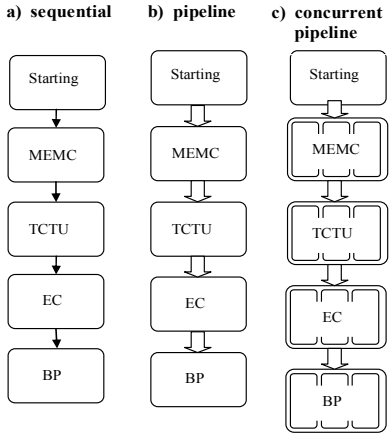

Figure 4. Concurrent structure.

All the application components have their own specific execution thread since they are defined as *RtUnit*s. However, the semantics specified in the communication media can constrain the final concurrency in the real implementation. All the connectors shown in Figure 1 are specified by a *CommunicationMedia* where the stereotype *CommSpecification* is applied. In Figure 4a), attributes *blockingFunctionCalls* and *blockingFunctionReturn* are valued as *true* which implies that the client application is blocked till its service request is dealt with and the service request finishes. This semantics implies that each application component is blocked till the application component called is finished, modeling simple function calls. In this manner, a concurrent structure is transformed into a sequential functionality.

Figure 4b) proposes a functional pipeline. The pipeline stages are identified by each application component. However, in this case, each application component should not be blocked by the execution of another application component. The corresponding communication media should have the attributes *blockingFunctionCalls* and *blockingFunctionReturn* values as false. In this way, each functional stage is not blocked till the finalization of the application called. Additionally, the attribute *srPoolSize* should be defined as 1, in order to constrain the number of threads available for dealing with the service request.

Finally, the case in Figure 4c) is a pipeline with internal concurrency. This design alternative applies a 3-way data-level

split in all functionalities. In order to model this communication semantics, the communication media should have the values of the attributes *blockingFunctionCalls* and *blockingFunctionReturn* as false. In this case, and in order to completely uncouple the application components, the communication media should be specified by the stereotype *StorageResource*, defining the *resMult* attribute with no value which enables the storing of function call requests. All the characteristics previously specified in the communication media mean that the concurrent structure modeled fits with Kahn process network semantics. The *srPoolSize* attribute should be specified as 3 as minimum value, in order to allow the generation of three threads to deal with the service request to avoid blocking the system.

Some additional semantics has to be considered in order to model the concurrent structure shown in Figure 4c). The proposed methodology enables interface inheritance. This inheritance allows the redefinition of the interfaces partitioning the sizes of the data streams sent and received. These streams are described in the model as parameters of these operations. As a consequence of splitting the data to be computed, three different, parallel execution flows can take place.

7. Conclusions and future work

The paper presents an approach for easy exploration of concurrency architectures in embedded designs. The approach presents both an extension to UML/MARTE to easily model the different architectures, and automatic synthesis of the SW communications from the UML/MARTE models. The automatic synthesis process enables easy exploration of different allocations of SW components, since simulators such as ISSs, virtualization tools and rapid prototyping can be used with low designer effort.

The system is initially described following the UML/MARTE standard plus additional channel semantics. The resulting model contains all information about functionality, concurrency and allocation required to perform the automatic synthesis. To do this, while maintaining sufficient simplicity in the visual diagrams, the information is displayed in several views. From this model, a generator synthesizes the communication wrappers completely ad-hoc for the application, reducing the overhead obtained with more generic design alternatives.

The approach enables easy design alternative specifications by focusing the system model on description of concurrency and synchronization within the channels. Additionally, the wrappers generated can implement different communications using basic communication facilities.

8. ACKNOWLEDGMENTS

This work has been founded by the PHARAON FP7-288307 and the Spanish TEC2011-28666-C04-02 MCI projects.

9. REFERENCES

[1] Y. Vanderperren, W. Mueller, and W. Dehaene, “UML for electronic systems design: a comprehensive overview,” Design Automation for Embedded Systems, 2008.

[2] L. Lavagno, G. Martin, B. Selic. “UML for real: design of embedded real-time systems”, ISBN 1-4020-7501-4.

[3] OMG: “UML profile for system on chip (SoC) specification” v1.0.1. 2006.

[4] "UML Profile for MARTE", www.omgmarte.org, 2009.

[5] S. Abdi, H. Yonghyun, Y. Lochi, C. Hansu, I. Viskic, D.D. Gajski. Embedded System Environment. “A Framework for TLM-based Design and Prototyping”. RSP, 2010.

[6] P.A. Hartmann, K. Gruttner, P. Ittershagen, A. Rettberg.”A framework for generic HW/SW communication using remote method invocation”. ESLSyn, 2011.

[7] F. Herrera, H. Posadas, P. Sánchez, & E. Villar, “Systematic Embedded Software Generation from SystemC”, DATE, 2003.

[8] D. Harel, H. Kugler, and A. Pnueli. “Synthesis revisited: Generating statechart models from scenario-based requirements”, Formal Methods in Software and System Modeling, 2005.

[9] S. Kang, H. Kim, J. Baik, H. Choi, C. Keum. ”Transformation Rules for Synthesis of UML Activity Diagram from Scenario-Based Specification”. COMPSAC, 2010.

[10] J. Barba, F. Rincón, F. Moya, J.D. Dondo J.C. López. “A comprehensive integration infrastructure for embedded system design”, Microprocessors and Microsystems, 2012.

[11] S. Zhenxin, W. Weng-Fai. “A UML-based approach for heterogeneous IP integration”. ASP-DAC, 2009.

[12] É. Piel, R. Atitallah, P. Marquet, S. Meftali, S. Niar, A. Etien, J.-L. Dekeyser, P. Boulet: "Gaspard2: from MARTE to SystemC Simulation", proc. of the DATE'08 workshop, 2008.

[13] V. Papailiopoulou, et al: “From design-time concurrency to effective implementation parallelism: The multi-clock reactive case”. Electronic System Level Synthesis Conference, 2011

[14] ISO/IEC 14496-2, “Coding of Audio-Visual Objects”, 2001.

[15] http://www.omg.org/spec/UML/2.4.1/

[16] OMG: "MDA guide, Version 1.1", June 2003.

[17] P. Peñil, H. Posadas, E.Villar. “Formal Modelling for UML-MARTE Concurrency Resources”. ICECCS, 2010.

[18] P. Peñil, J. Medina, H. Posadas, E. Villar. “Generating heterogeneous executable specifications in SystemC from UML/MARTE models”. Innovations in System Software Engineering, 2010.

[19] C. Mraidha et all, Y. Tanguy, C. Jouvray, F. Terrier, S. Gérard. “An Execution Framework for MARTE-based Models”, ICECCS 2008.

[20] F.Herrera and E.Villar “A Framework for Embedded System Specification under Different Models of Computation in SystemC”, Design Automation Conference. 2006

[21] QEMU, www.qemu.org

[22] Open Virtual Platforms, http://www.ovpworld.org/

[23] Multicube Explorer, http://home.dei.polimi.it/zaccaria/multicube_explorer_v1/Home.html

[24] Zeligsoft, www.zeligsoft.com

Automatic Transformation of Abstract AUTOSAR Architectures to Timed Automata

Stefan Neumann, Norman Kluge and Sebastian Wätzoldt
Hasso-Plattner Institute at the University of Potsdam, Germany
stefan.neumann|sebastian.waetzoldt@hpi.uni-potsdam.de
norman.kluge@student.hpi.uni-potsdam.de

ABSTRACT

The AUTomotive Open System ARchitecture (AUTOSAR) is the emerging standard for the development of real-time embedded automotive systems. Several tools exist that support the development as well as the analysis of AUTOSAR systems. Simulation environments use models or generated source code for testing and scenario-based simulation purposes. Unfortunately, there is a lack of methods and tools supporting the early timing analysis of AUTOSAR systems. In this work, we show how to automatically transform a given AUTOSAR architecture to an interconnected set of timed automata that represents the state-based timing behavior of the system. The derived timed automata models are used for analyzing the timing behavior in an early development stage. Furthermore, we show how to analyze the resulting timing behavior supporting abstract and incomplete AUTOSAR systems using the tool UPPAAL.

1. INTRODUCTION

AUTOSAR is the upcoming standard supporting a distributed development process of complex embedded automotive real-time systems. Several tools support the specification as well as analysis of AUTOSAR conform systems. In the embedded domain, testing and simulation play an important role for system verification [4]. Analysis tools like SystemDesk[1] provide simulation capabilities for validating the functional behavior of an AUTOSAR conform system. Beneath the functional behavior, non-functional properties as in the case of timing are crucial for automotive embedded systems. Unfortunately, there is a lack of methods and tools supporting the analysis of non-functional properties as timing. In this work, we show how to automatically derive an interconnected set of timed automata based on a given AUTOSAR architecture. The automatic derivation includes the most relevant aspects that allow us to apply a timing analysis using elaborated tools as UPPAAL (cf. [3]). Furthermore, abstraction is supported in such a manner that also a partially defined system can be analyzed, e.g., for deciding if local deadlines respectively periods of operating system tasks are met. Thus, the distributed development of such architectures is supported allowing an early analysis of the partially defined AUTOSAR architecture. The paper is organized as follows: We give a brief introduction into the AUTOSAR architecture in Sec. 2 as well as into timed automata in Sec. 3. In Sec. 4, we schematically describe how to automatically derive a set of timed automata based on a given AUTOSAR description. In Sec. 5, we discuss an application example and show how to use the derived model for realizing an analysis also in case of an incomplete system. We conclude the paper with a discussion about related work in Sec. 6.

[1]Provided by dSPACE (see `www.dspace.com`).

2. AUTOSAR

The AUTomotive Open System ARchitecture is a framework for the development of real-time embedded automotive systems. AUTOSAR provides a layered architecture consisting of the software layer (SW), the runtime environment (RTE), the basic software layer (BSW) and the hardware layer (HW) as shown on the left of Fig. 1. At the top layer, software components (SWCs) (① and ②) are the building blocks for realizing the behavior of the overall developed application. Communication between individual SWCs is realized via ports ② and so-called assembly connectors ③. While ports are part of the SW layer, communication between ports via connectors is realized by the RTE. Additionally, the component-based development methodology in AUTOSAR at the top layer supports compositions, which group an individual set of SWCs together. Within a composition, delegation connectors transfer the data of the internal ports of SWCs to external ports of the composition.[2] Because compositions are a set of grouped SWCs and we can derive TA from SWCs as well as realize the assembly connectors, we focus on SWCs and do not discuss compositions in more detail in the reminder of this work. The behavior of a SWC is realized by so-called Runnables ① that are associated with an implementation. Such an implementation can exist in form of C functions. Runnables are able to interact with ports ② of the SWC by reading or writing data from resp. to it. The RTE handles the communication between different constituents of the application layer, e.g., different SWCs, and between the application layer and the BSW layer, e.g., for accessing HW.[3]

[2]See: Specification of the Virtual Functional Bus - V. 1.2.0.
[3]See: General Requirements on BSW Modules - V. 3.2.0.

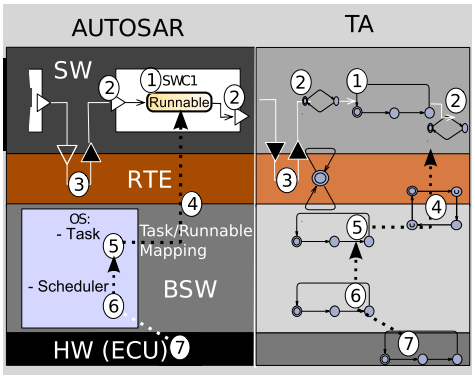

Figure 1: The AUTOSAR layered architecture.

Below the RTE the BSW layer provides services, e.g., in form of drivers for accessing the HW as well as the operating system (OS) functionalities.[4] The OS needs a description ④ of the execution order of the Runnables mapped on a task ⑤. In the remainder of this work, such a description of the execution order is called the Task-Runnable mapping (TRM). At runtime, an OS scheduler ⑥[5] handles the execution of each task according to its configuration, e.g., in form of the defined period and priority while using the resources of the HW ⑦. The TRM further defines, in which cases the mapped Runnables are triggered. As an example, it is possible to execute an OS task with a period of 100 ms. The TRM allows to define that one of the mapped Runnables has to be triggered only each 200 ms or in other words each second period of the task.

In the reminder of this work, we show how to automatically derive from a given AUTOSAR architecture an interconnected set of timed automata that allows us to analyze the resulting timing behavior including the previously mentioned elements. We show how the derived automata reflect the control flow as well as the communication. The control flow exists in form of the dependencies between the scheduler using the resources of the HW, OS tasks as well as Runnables like depicted by the dashed lines on the left of Fig. 1. The communication exists in case of sent and received signals delegated between different SWCs via the RTE as well as signals that are read or written by Runnables from and to ports like depicted by the solid lines on the left of Fig. 1. Furthermore, the figure schematically shows the derived timed automata on the right for each element.

3. UPPAAL TIMED AUTOMATA

In the following, we recall Alur-Dill style timed automata (TA) like they are used in the tool UPPAAL. A schematic description of an UPPAAL TA is depicted in Fig. 2. Such a TA A can be understood as a 6 tuple $A = (\Sigma, \mathcal{L}, \mathcal{L}^0, X, \mathcal{I}, E)$ where Σ represents the set of signals, respectively channels (e.g., signal **{t}Run** in Fig. 2), $\mathcal{L}$ represents the set of locations (e.g., **ready**), $\mathcal{L}^0$ the initial location, $\mathcal{I}$ a set of invariants assigned to locations (e.g., $x <= \{p\}$ assigned to location **ready** with $p \in \mathbb{N}^+$) and E the set of edges (e.g., the edge **e**). An edge **e** can contain signals that need to be sent (!) or received (?) to take an edge, guards (e.g., $x == \{p\}$ for edge **e**) that need to be fulfilled to enable the edge and variable assignments as in the case of the clock reset $x := 0$ of the edge **e**. Thus, the edge **e** leading to the initial location can be taken if the guard $x == \{p\}$ is fulfilled, resetting the clock x to 0. In such a manner, periodic behavior, e.g., of an OS task can be modeled using TA. If a signal is included in an edge, another TA is required to send respectively receive the signal to take the edge. Furthermore, in the used model of TA *urgent* and *committed* locations are used. If a TA is in an urgent location Ⓤ time is not allowed to pass. A committed location ⓒ (e.g., Fig. 2 location **error**) is a strengthened form of urgent locations, where time is not allowed to pass. When being in a committed location only edges are enabled leaving a committed location. A state of a TA results from the current location in combination with the assignment of variables (clocks and discrete state information). In this work, we do not focus on the exact semantics of the formal model of TA. For a more detailed description as well as for more information about UPPAAL see [2, 3].

[4]See: Specification of Operating System - V. 5.0.0.
[5]See: Specification of OS, 2011, V. 5.0.0

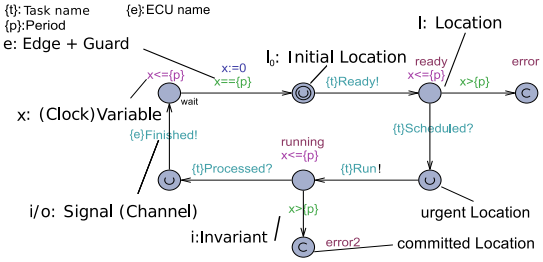

Figure 2: Schematic description of a UPPAAL TA realizing the behavior of an OS task.

4. TRANSFORMATION

In this section, we show how the previously introduced constituent parts of the AUTOSAR architecture are realized by different TA pattern, resp. templates. These templates are used to describe the resulting structure of each TA, representing an individual constituent part of a given AUTOSAR architecture. The goal of each template is to represent the most abstract behavior of each constituent part. The real TA are later on instantiated by a java program according to these templates.

4.1 Components

At first, we start with SWCs consisting of the parts as schematically depicted on the top left of Fig. 1. We use templates for the representation of Runnables ① respectively their implementation, ports ② as well as the Task-Runnable mapping ④.[6]

Runnables/Implementations

For each Runnable, we derive one automaton including the abstract representation of the behavior[7] like best-case and worst-case execution times (if available) as well as the interaction with the ports (reading and writing data elements).

[6]Normally ④ is part of the RTE, but for a better understanding this part is discussed in the context of SWCs.
[7]At this point, we hide the implementation details to protect intellectual properties.

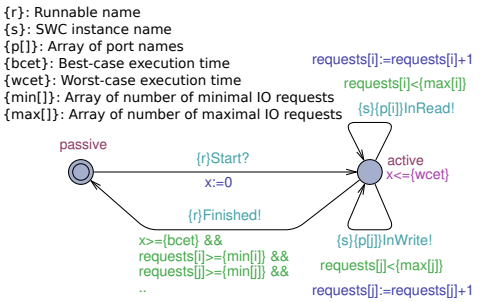

Figure 3: Schematic description of an UPPAAL TA realizing the behavior of a Runnable implementation ①.

Furthermore, a second TA is derived that is responsible for connecting the Runnable instance with the task, on which it is mapped ④. Because implementations of Runnables can vary in a wide range, this variability is also reflected in the template used for the Runnable behavior respectively its implementation. Fig. 3 shows the template of a Runnable **r** contained in the SWC identified by its instance name **s**. As a convention, all braces {} that are surrounding, e.g., instance names, are later on replaced with the concrete instance values. E.g, {*s*} is later on replaces by the name of the SWC. In its basic form, the template automaton contains two locations. The initial location (**passive**) is associated with the state, in which the Runnable is not executed and the second location (**active**) is associated with the state, in which it is executed. Transitions between the **passive** and **active** location are only possible by receiving an activation signal via the channel **{r}Start** in the initial location respectively by sending a completion signal via the channel **{r}Finished** after execution (sent, resp. received from the TA shown in Fig. 4). In the location **active**, the Runnable can read and write on variables, e.g., internal ones or variables associated with ports of the surrounding SWC ③. If no information about the state-based behavior of the implementation is given, read and write accesses to data can occur randomly in the active location. The ports of the SWC are accessed by sending or receiving the appropriate signals. Individual ports are identified via the SWC instance name **s** in combination with the array **p**[], containing all port names of the SWCs. If available, the minimal (**min**[]) an maximal (**max**[]) number of sent and received signals, allowed to be sent during one execution of the Runnable, can be specified for each individual port.[8] In the passive location, data access can obviously not occur due to the fact that writing or reading variable values requires the resource of the CPU, which is in the initial location not allocated for the Runnable (the Runnable is not executed). Furthermore, the template allows specifying best-case (BCET) and worst-case (WCET) execution times. Values for BCET and WCET can potentially be derived from different sources, e.g., using WCET analyzers, measured values known from previous projects or estimated values. In the template, the transition from location **active** to **passive** can only take place in the case the clock **x** has a value between BCET and WCET. The signal **{r}Start** needs to be sent to execute the Runnable. Executing a Runnable is realized by a task but in the AUTOSAR

[8]For each signal an individual edge is created in the resulting TA based on the description of the template.

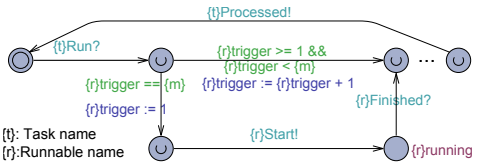

Figure 4: Schematic description of a UPPAAL TA realizing the Task-Runnable mapping ④.

standard, tasks do not directly trigger Runnables. Instead, the TRM ④ is responsible for sending this signal. The TRM is part of the RTE configuration and realizes the appropriate execution of Runnables in the context of an OS task. It defines the execution order as well as the conditions, under which each mapped Runnable is executed. An excerpt of the TA template realizing the TRM is shown in Fig. 4. If a task **t** is executed by the scheduler a signal **{t}Run** is sent from the task (see template shown in Fig. 2) to the TRM. In the context of the TRM TA, the appropriate Runnables are triggered in a defined order, e.g., starting with the Runnable mapped on the associated SWC with name **r**, listening on the signal **{r}Start**. Depending on the logic specified by the mapping, different Runnables are triggered, allowing realizing alternating execution orders of Runnables or realizing the conditional execution of Runnables. Due to space limitations, we only discuss one example of a mechanism provided by the TRM in case of AUTOSAR timing events. Timing events in AUTOSAR can be used to specify the period of a Runnable. It can be a multiple of the period from the corresponding task. Consider an OS task **t** with a period **p**. The TRM can be configured such that a Runnable is triggered only each **m** executions of the task **t**. Thus, the Runnable is executed with a period equal to $\mathbf{p} * \mathbf{m}$. For realizing the appropriate behavior of timing events using a TA, for each Runnable **r**, a counter variable **{r}trigger** is created inside the TRM. After **m** executions of the task, the Runnable will be triggered and the counter is reset. Fig. 4 shows a schematic description of the TRM realizing the behavior of timing events. If no timing event exists for a Runnable **r**, the associated variable **m** is initialized with the value 0. According to the given configuration of the AUTOSAR architecture, a construct as depicted in Fig. 4 is generated for every Runnable and inserted at the appropriate position instead of the three dots currently shown. The signal **{t}Processed** is sent by the TA realizing the TRM to the OS task (see Fig. 2) after all Runnables were processed, indicating the completion of the Runnables mapped on the task.

Ports

The next constituent part, required for representing the state-based timing behavior of a SWC, is the realization of SWC ports ③. In AUTOSAR, ports provide access to SWCs respectively allow SWCs to access other connected components. Because variables of ports are always written and read within the execution context of a Runnable respectively the associated implementation, the ports itself do not consume time when being read or written and simply forward signals. So in our model, the time for reading or writing from or to a port is added to the execution context of the Runnables. As a result, ports are modeled like shown in Fig. 5, where for each port **p** of a SWC **s** a TA is created.

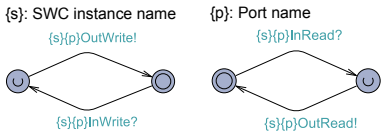

Figure 5: Schematic description of a UPPAAL TA realizing the behavior of the ports of a SWC.

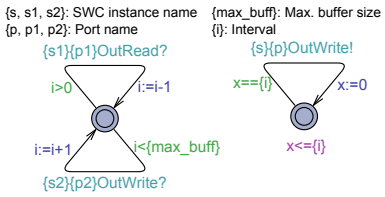

Figure 6: Schematic description realizing the communication between SWCs in form of a connector.

The TA on the left of Fig. 5 shows the template realizing an output port and the TA on the right realizes an input port.

4.2 Runtime Environment

In the following, we consider the communication between different SWCs. The RTE realizes communication between SWCs in AUTOSAR ③. Connectors link SWCs and the RTE is responsible for realizing this communication link. Again, as in the case of ports, connectors do no consume time. The time for delegating signals between SWCs is always added to the execution time of Runnables. On the left of Fig. 6, the behavior of a connector is schematically depicted. Within this template connected ports of different SWCs simply forward signals. Specifying the mapped SWCs, the parameter **s1, s2, p1, p2** need to be defined such that **s1** is the identifier of the receiving SWC, **s2** is the identifier of the sending SWC, **p1** is the identifier of the receiving SWC port and **p2** is the identifier of the sending SWC port.

Furthermore, AUTOSAR supports different communication mechanism as in the case of the *last value best value* semantic, where not consumed signals are simple overwritten. Alternatively, buffered communication can be used, where signals are stored and a buffer overflow or an empty buffer can lead to a failure. The template shown in Fig. 6 realizes such a behavior by only allowing sending a signal via a connector if not already **max-buff** signals are stored. Reading a signal from a connector is only possible if the buffer is not empty. For this purpose the integer variable **i**, appropriate guards as well as updates are added to the edges of the template. In case of a not buffered communication these variables, guards and updates are simply removed from the template.[9]

4.3 Operating System

In the context of the BSW, we focus on two relevant OS parts, which have major impact on the resulting timing behavior of the specified AUTOSAR systems. Subsequently, we show how to model templates for the OS tasks and the OS scheduler.

[9]In AUTOSAR further communication mechanisms are supported not considered here due to space limitations.

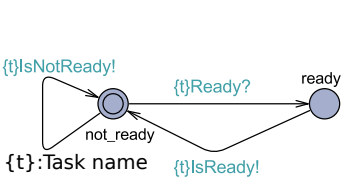

Figure 7: Schematic description of a UPPAAL TA representing the state of a task.

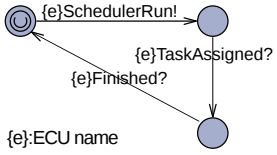

Figure 8: Schematic description of a UPPAAL TA realizing the behavior of the CPU.

OS Tasks

The task template shown in Fig. 2 represents an OS task with name **t**. When being in the initial urgent location the signal **{t}Ready** is sent without allowing time to pass to indicate that the task is ready to be executed or in other words in the ready state. The current state of each task is stored in an additional TA allowing keeping the task template more compact by using two templates instead of one. The template that receives the signal **{t}Ready** and stores the state (**not_ready** or **ready**[10]) is shown in Fig. 7. The scheduler is able to check if a task **t** is currently in the state **ready** by sending the signal **{t}isReady** to the TA storing this information about task **t** (see Fig. 7). After the task has sent the signal **{t}Ready**, the task template changes to the successor location, where it is allowed to stay for at most the time till its period **p**, measured by the clock **x**, is expired. If the value of the clock **x** becomes larger than **p**, the task switches to a location indicating that the task has not been executed before its period. In the case task **t** is in location **ready**, the scheduler can activate this task by sending the signal **{t}Scheduled**. If this signal is received from the scheduler by the task template, the task changes to the location **running** without allowing time to pass while sending the signal **{t}Run** to the TA realizing the TRM like depicted in Fig. 4. The TRM is responding with the signal **{t}Processed** if all Runnables have been executed. If this signal has not arrived before the period of the task is over, again, the task template switches to a location that indicates an error. After signal **{t}Processed** has been sent, the signal **{e}Finished** is sent to the CPU (see Sec. 4.4) without allowing time to pass. Afterwards, the CPU activates the scheduler, which chooses the next task to execute. The task template takes a transition back to the initial location in the case its period is over, resetting the clock x to the value 0.

Scheduler

The template of the scheduler (see Fig. 9) implements a rate monotonic scheduler (for more information about rate monotonic scheduling cf. [5]).[11] The scheduler is triggered via the signal **{e}SchedulerRun** sent by the CPU **e** on which it is executed. By receiving the signal **{e}SchedulerRun**, the scheduler changes to the first successor location trying to activate a task via synchronizing with the signal **{t[0]}isReady**, resp. **{t[0]}isNotReady** in case the task is not ready. These signals are sent by the appropriate TA representing

[10]Because we only consider rate monotonic scheduling with non preemptive tasks, these two states of a task are sufficient.

[11]For more information about stopwatches (supported by UPPAAL) and preemptive scheduling see [1].

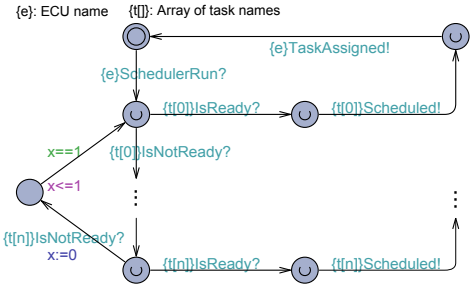

Figure 9: Schematic description of a UPPAAL TA realizing the behavior of the scheduler.

the state of each task (see Fig. 7). The array **{t[i]}** contains all tasks ordered by their priority, e.g., containing at the index $i = 0$ the task with the highest priority. Accordingly, tasks with a higher priority are activated first, if they are in the state **ready** (cf. Fig. 7). After sending the signal **{t[i]}Scheduled** to this task (see Fig. 2), the scheduler sends the signal **{e}TaskAssigned** to the template realizing the resource of the CPU (see Fig. 8), indicating that the CPU now is allocated by the task. Afterwards, the scheduler switches back to the initial location. If no task contained in **t[]** is ready the TA realizing the scheduler enters the location on the left of Fig. 9. In this location the scheduler is waiting for one time unit (assuming that one time unit is smaller or equal to the smallest tick of the OS and thus no task becomes ready before one time unit) before checking the tasks for readiness again.

4.4 CPU

The HW is reflected in form of the CPU ⑦. We only consider HW containing a single core. The template realizing the behavior of the CPU is shown in Fig. 8. The CPU has three different locations, the initial one, where the CPU is not occupied and being in some idle state, the second, where the scheduler is running and the third, where a task is executed. When being in the initial location the scheduler is triggered via sending the signal **{e}SchedulerRun**. When the template of the ECU is in the initial location, this signal is sent without allowing time to pass (the initial location is an urgent location). When being in the second location and the scheduler activates a task, signal **{e}TaskAssigned** is sent from the scheduler to the ECU. If a running task sends the signal **{e}Finished** to the CPU, the CPU template enters the initial location, triggering the scheduler again for choosing the next task to activate.

5. ANALYSIS

As an evaluation example, Figure 10 shows the composition of the three SWCs *FuelSensor*, *FuelController* and *EngineModel* realizing the fuel rate control of a combustion engine.[12] SWC *EngineModel* represents relevant behavior of the engine, required for validating the overall functionality. It receives the desired fuel rate from component *FuelsysController* and sends raw values to the component *FuelSensor*. SWC *EngineModel* contains a single Runnable *Engine*, mapped to a task *TaskEngine* that is executed with a period of 5 ms. *FuelSensor* is responsible for evaluating and preprocessing raw values from the engine. It contains Runnable *DetectSensorFailures*, responsible for checking raw input values (if they are out of range), and Runnable *SensorCorrection*, responsible for deriving roughly corrected values, if required. Both Runnables are mapped onto the single task *TaskFuelSensor*, executed with a period of 10 ms. SWC *FuelController* is responsible for calculating the desired fuelrate for the engine base on the corrected sensor values. SWC *FuelController* consists of the two Runnables *AirFlowCalculation* and *FuelRateCalculation*, both mapped on task *TaskFuelConroller*, which is also executed with a period of 10 ms. The resulting system is transformed to a set of TA according to the templates previously described. We applied model checking using UPPAAL by searching for deadlocks. Searching deadlocks requires exploring the full reachable statespace and as a result is a valuable test for investigating the complexity of the system. Table 1 shows the number of TA (#TA) contained in the derived system as well as the required verification time and explored states (according to the UPPAAL console tool). For the evaluation example no deadlock was discovered and the complexity seems to be rather small. We also applied the overall procedure to the original version of the application example shipped with professional tool SystemDesk, which contains three more Runnables as well as an additional port between SWCs *FuelSensor* and *FuelController*. As a third example we used a more complex application of a turn-light control, also provided by the tool SystemDesk. This example consists of 5 tasks, 5 SWCs and 8 Runnables. In the last example a deadlock occurs, due to the fact that a period of a task is missed (see Sec. 5.2 for checking missed periods). The result is shown in Table 1. It can be seen that the complexity is moderate. As future work more complex examples need to be evaluated to gain more significant results.

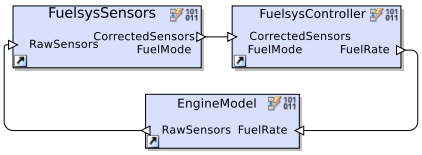

Figure 10: Application example - Engine Control

Table 1: Application name, contained number of TA and verification results (required time (seconds) and space (states)).

System	#TA	Time/States
Fuel-Control	27	0.21/3988
Fuel-Control (original)	31	0.25/4149
Turn-Light	45	0.8/7146

5.1 Abstract and Incomplete Behavior

When developing an AUTOSAR conform system there is often the situation that not all relevant system parts are available (e.g., still in development or because stakeholder are not willing to provide detailed information). As a consequence, often, only an incomplete system is available. When analyzing such an incomplete system via a set of derived TA, the resulting observable timing behavior may be dif-

[12]The AUTOSAR example is derived from an existing demo application shipped with the tool SystemDesk (see http://www.dspace.com/systemdesk).

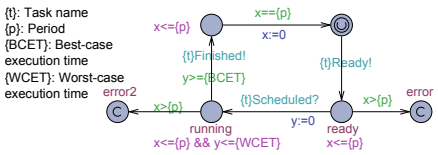

Figure 11: Schematic description of a UPPAAL TA realizing a task simulating a part of the remaining system.

ferent from the timing behavior of the later system. For example, other high priority tasks, which are not available or which are not considered till the current development stage, can potentially lead to significant differences for the timing behavior. Fig. 11 depicts an OS task with period **p** as well as **BCET** and **WCET** execution times. Such tasks can be used to simulate the resource consumption of the CPU. As in the task template in Fig. 2, if the task period is missed, an error location will be reached. The TA template in Fig. 11 contains the same states as the task template without the two urgent locations (without synchronization with the TRM). Therefore, we support the analysis of abstract and incomplete system descriptions at the level of the OS. An open point is how to support unconnected ports of SWCs. Especially, buffered communication of unconnected ports leads in the absence of signals to empty buffers. The right TA template in Fig. 6 sends signals to an unconnected port. It is a simplified version of the left TA in Fig. 6 as described in Sec. 4.2. The template on the right allows to define a period with the signals **{s}{p}OutWrite**, which are sent to the open port of a SWC **s**. The period is defined by the used guard of the edge in combination with the invariant. As a result, each i time units a signal is sent. Potentially, more elaborated templates can be used defining upper and lower bound or timer intervals, in which signals are sent.

5.2 Checking Properties

Properties are another open point that can be analyzed using the derived TA model. An example of such a property is a given period of OS tasks, which must be checked if it always holds. For this purpose, the OS task template (Fig. 2) includes an outgoing edge of the locations **running** and **ready** that lead to an error location. Therefore, we cover both cases, where we reach an error state. First, the period of the task has passed before the task is triggered or second, the execution of the mapped Runnables has finished to late. In UPPAAL, state formulas can be used to query the reachability of error states and checked if certain properties hold. Potentially, other error locations can be added depending on the properties that need to be analyzed.

We have implemented a Java prototype, which generates the UPPAAL TA according to the above described templates from an AUTOSAR conform XML description. Furthermore, we support additional AUTOSAR events and interruptible OS tasks.[13]

6. CONCLUSION AND RELATED WORK

In this work, we have shown how to automatically transform a given AUTOSAR architecture to the formal model of TA. With the set of TA, we apply simulation and verification analysis techniques. Partial information about the later implementation (BCET and WCET) are sufficient for an early timing analysis. Furthermore, the introduced approach supports the analysis of the timing behavior of an incomplete AUTOSAR architecture. The approach in [9] uses the SymTA/S tool to apply a real-time analysis of an AUTOSAR architecture. In this approach, the overall system needs to exist. Therefore, the considered parts of the system need to be defined or abstracted for achieving gray-box models for the blanked parts. However, how to achieve gray-box models is not described. In contrast our approach allows starting with a fraction of a system that is not already fully specified. Instead of deriving abstract descriptions based on more concrete ones, the system parts can be partially defined and afterwards refined. In [8], they use SystemC (cf. [6]) for the behavior specification of an AUTOSAR architecture reflecting timing properties. In contrast to this work, the SystemC model needs to be defined manually what requires significant additional effort. The approach in [7] automatically derives a TA model based on a given SystemC description. Again, it requires the complete SystemC models, which must be specified first. In our approach, we only use artifacts, in form of the given AUTOSAR architecture description that need to be created in any case when building an AUTOSAR conform system.

[13]These additional constructs are not discussed in this work due to space limitations.

References

[1] Y. Abdeddaïm and O. Maler. Preemptive Job-Shop Scheduling Using Stopwatch Automata. In *TACAS'02*, volume 2280 of *LNCS*, pages 39–53. Springer, 2002.

[2] R. Alur and D. L. Dill. A Theory of Timed Automata. In *Theoretical Computer Science*, volume 126. Elsevier Science Publishers Ltd., 1994.

[3] G. Behrmann, A. David, and K. Larsen. A Tutorial on Uppaal. In *SFM-04:RT*, LNCS. Springer, 2004.

[4] B. Broekman and E. Notenboom. *Testing Embedded Software*. Wesley, 2003.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 1 edition, 2004.

[6] A. Donlin, A. Braun, and A. Rose. Systemc for the design and modeling of programmable systems. In *FLP'04*, volume 3203 of *LNCS*. Springer Berlin / Heidelberg, 2004.

[7] P. Herber, M. Pockrandt, and S. Glesner. Automated Conformance Evaluation of SystemC Designs using Timed Automata. In *ETS'10*. IEEE Computer Society, 2010.

[8] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosentiel. Timing simulation of interconnected AUTOSAR software-components. In *DATE '07*. EDA Consortium, 2007.

[9] K. Richter, N. Feiertag, M. Rudorfer, O. Scheickl, and C. Ainhauser. How Timing Interfaces in AUTOSAR can Improve Distributed Development of Real-Time Software. In *GI Jahrestagung (2)*, pages 662–667, 2008.