Workshop Proceedings

# ACES$^{MB}$ 2011

## Fourth International Workshop on Model Based Architecting and Construction of Embedded Systems

October 18th, 2011, Wellington, New Zealand

Organized in conjunction with MoDELS 2011
14th International Conference on Model Driven Engineering Languages and Systems

Edited by:
Stefan Van Baelen (K.U.Leuven - DistriNet, Belgium)
Sébastien Gérard (CEA - LIST, France)
Ileana Ober (University of Toulouse - IRIT, France)
Thomas Weigert (Missouri University of Science and Technology, USA)
Huascar Espinoza (ESI Tecnalia, Spain)
Iulian Ober (University of Toulouse - IRIT, France)

# Table of Contents

# Foreword

The development of embedded systems with real-time and other critical constraints raises distinctive problems. In particular, development teams have to make very specific architectural choices and handle key non-functional constraints related to, for example, real-time deadlines and to platform parameters like energy consumption or memory footprint. The last few years have seen an increased interest in using model-based engineering (MBE) techniques to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models in a layered construction of systems. MBE techniques are interesting and promising for the following reasons: They allow to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models, and they support a layered construction of systems, in which the (platform independent) functional aspects are kept separate from architectural and non-functional (platform specific) aspects, where the final system is obtained by combining these aspects later using model transformations.

The objective of this workshop is to bring together researchers and practitioners interested in model-based engineering to explore the frontiers of architecting and construction of embedded systems. We are seeking contributions relating to this subject at different levels, from modelling languages and semantics to concrete application experiments, from model analysis techniques to model-based implementation and deployment. Given the criticality of the application domain, we particularly focus on model-based approaches yielding efficient and provably correct designs. Concerning models and languages, we welcome contributions presenting novel modelling approaches as well as contributions evaluating existing ones. The workshop targets in particular:

- Architecture description languages (ADLs). Architecture models are crucial elements in system and software development, as they capture the earliest decisions which have a huge impact on the realisation of the (non-functional) requirements, the remaining development of the system or software, and its deployment. We are particularly interested in examining:
  - Position of ADLs in an MBE approach;
  - Relations between architecture models and other types of models used during requirement engineering (e.g., SysML, EAST-ADL, AADL), design (e.g., UML), etc.;
  - Techniques for deriving architecture models from requirements, and deriving high-level design models from architecture models;
  - Verification and early validation using architecture models.

- Domain specific design and implementation languages. To achieve the high confidence levels required for critical embedded systems through analytical methods, in practice languages with particularly well-behaved semantics are often used, such as synchronous languages and models (Lustre/SCADE, Signal/Polychrony, Esterel), super-synchronous models (TTA, Giotto), scheduling-friendly models (HRT-UML, Ada Ravenscar), or the like. We are interested in examining the model-oriented counterparts of such languages, together with the related analysis and development methods.
- Languages for capturing non-functional constraints (MARTE, AADL, OMEGA, etc.)
- Component languages and system description languages (SysML, MARTE, EAST-ADL, AADL, BIP, FRACTAL, Ptolemy, etc.).

We accepted 6 full papers for the workshop. We hope that the contributions for the workshop and the discussions during the workshop will help to contribute and provide interesting new insights in Model Based Architecting and Construction of Embedded Systems.

The ACES$^{MB}$ 2011 organising committee,

Stefan Van Baelen,
Sébastien Gérard,
Ileana Ober,
Thomas Weigert,
Huascar Espinoza,
Iulian Ober.

The ACES$^{MB}$ 2011 steering committee,

Mamoun Filali,
Susanne Graf.

October 2011.

# Acknowledgments

The Organizing Committee of ACES$^{MB}$ 2011 would like to thank the workshop Programme Committee for their helpful reviews.

Jean-Michel Bruel (University of Toulouse-IRIT, France)
Agusti Canals (CS-SI, France)
Daniela Cancila (Atego, France)
Arnaud Cuccuru (CEA-LIST, France)
Huascar Espinoza (ESI Tecnalia, Spain)
Jean-Marie Farines (UFSC, Brasil)
Mamoun Filali (University of Toulouse-CNRS-IRIT, France)
Robert France (CSU, USA)
Pierre Gaufillet (Airbus, France)
Sébastien Gérard (CEA LIST, France)
Susanne Graf (Univ. Joseph Fourier-CNRS-VERIMAG, France)
Bruce Lewis (US Army, USA)
Ileana Ober (University of Toulouse-IRIT, France)
Iulian Ober (University of Toulouse-IRIT, France)
Isabelle Perseil (Inserm, France)
Dorina Petriu (Carleton University, Canada)
Andreas Prinz (University of Agder, Norway)
Bernhard Rumpe (RWTH Aachen, Germany)
Douglas C. Schmidt (Vanderbilt University, USA)
Bran Selic (Malina Software, Canada)
Martin Törngren (KTH, Sweden)
Stefan Van Baelen (K.U.Leuven DistriNet, Belgium)
Tullio Vardanega (University of Padua, Italy)
Eugenio Villar (Universidad de Cantabria, Spain)
Thomas Weigert (Missouri S&T, USA)
Tim Weilkiens (OOSE, Germany)
Sergio Yovine (VERIMAG, France)

This workshop is organised as an event in the context of

- The IST-004527 ARTIST2 Network of Excellence on Embedded Systems Design
- The research project EUREKA-ITEA2 EVOLVE (Evolutionary Validation, Verification and Certification)
- The research project EUREKA-ITEA2 VERDE (Validation-driven design for component-based architectures)
- The research project EUREKA-ITEA2 OPEES (Open Platform for the Engineering of Embedded Systems)

# Towards Integrated System and Software Modeling for Embedded Systems

Hassan Gomaa

Department of Computer Science
George Mason University, Fairfax, VA
hgomaa@gmu.edu

**Abstract.** This paper addresses the integration of system modeling and software modeling, particularly for embedded systems, which are software intensive systems that consist of both hardware and software components. This paper describes a systems modeling approach to create structural and behavioral models of the total system using SysML. The systematic transition to software modeling using UML is then described.

**Keywords**: systems modeling, software modeling, SysML, UML, embedded systems, structural modeling, behavioral modeling.

## 1 Introduction

Model-based systems engineering [1, 2] and model-based software engineering [3, 4, 5, 6] are increasingly recognized as important engineering disciplines in which the system under development is modeled and analyzed prior to implementation. In particular, embedded systems, which are software intensive systems consisting of both hardware and software components, benefit considerably from a combined approach that uses both system and software modeling. This paper describes a modeling solution to this problem with an approach that integrates these two disciplines for the development of embedded systems. In particular this paper concentrates on developing the hardware/software boundary of a system, the decomposition of the system into hardware and software components, and designing the interface between hardware and software components. The modeling languages used in this paper are SysML [7] for systems modeling and UML [8, 9] for software modeling.

This paper describes a systems modeling approach to develop a multi-view model of the system, in terms of a structural block definition diagram of the problem domain, a system context block definition diagram, a use case model, and a state machine model, which forms the basis for a transition to software models. This is followed by modeling the hardware/software boundary, which involves decomposing the system into hardware and software components and modeling the possible deployment of components. The steps in software modeling that address the software side of the hardware/software interface are described next, in which the boundary of the software system is established and the software components are determined. From there, the software components are categorized as active (i.e., concurrent) or passive and their behavioral characteristics are determined. Although the approach can be used for most systems, it is intended in particular for embedded systems, which are software intensive systems that typically have a large number of hardware components, including sensors and actuators, which need corresponding software components to interface and communicate with them. An example of this model-based approach is given using a Microwave Oven system.

## 2 System and Software Modeling

The objective of the model-based approach described in this paper is to clearly delineate between systems modeling and software modeling, with a well-defined transition between the two phases. This section describes the steps in systems modeling, hardware/software boundary modeling, and software modeling.

### 2.1. Overview of System Modeling

System modeling consists of structural and behavioral (dynamic) modeling of the total system using SysML to get a better understanding of the system. The following steps consider the total system perspective, consisting of hardware, software and people, without consideration of what functionality is carried out in hardware and what functionality in software.

1. Structural modeling of the problem domain using block definition diagrams. In structural modeling of the problem domain, the emphasis is on modeling real-world entities, including relevant systems, users, physical entities and information entities.
2. System context modeling. A system context block definition diagram explicitly shows the boundary between the total system, which is treated as a black box, and the external environment. In considering the total hardware/software system, users and external systems are external to the system, while hardware and software entities are inside the system.
3. Use case modeling. In order to get an understanding of the system behavior, use case modeling is carried out. This involves determining the actors (users) of the system and the sequence of interactions between the actor(s) and the system.
4. Dynamic state machine modeling. State machines provide a more precise method for modeling the behavior of state-dependent embedded systems. For these systems, state machine modeling is preferred to activity diagrams because embedded systems are highly state dependent.

### 2.2. Overview of Hardware/Software Boundary Modeling

The following steps address the decomposition of the total system into hardware and software components, in particular determining what is done by hardware and what is done by software.

1. Decompose system into hardware and software components. Develop a block definition diagram that depicts the hardware components and the software system.
2. Deployment modeling. Develop deployment diagram depicting the deployment of hardware and software components.

### 2.3 Overview of Software Modeling

Once the hardware/software boundary has been determined, the next steps address the decomposition of the software system into its constituent components. They involve determining the boundary of the software system, and determining those software components that interface to and communicate with the hardware components.

1. Software context modeling. Unlike system context modeling, software context modeling depicts the boundary of the software system with the hardware components external to the software system.
2. Software component structuring. This step involves determining the software components that are needed to interface to and communicate with the hardware components. The software components are further categorized as active (concurrent) or passive.
3. Having determined the boundary between the hardware and software components, the subsequent development steps follow a UML-based software modeling and design method, in particular the COMET [4] method.

## 3. System Modeling

### 3.1 Structural Modeling of the Problem Domain

In structural modeling of the problem domain, the initial emphasis is on modeling real-world entities, including relevant systems, users, physical entities and information entities. Physical entities have physical characteristics – that is, they can be seen and touched. Such entities include physical devices, which are often part of the problem domain in embedded applications. Information entities are conceptual data-intensive entities that are often persistent – that is, long-living. Information entities are particularly prevalent in information systems (e.g., in a banking application, examples include accounts and transactions). Structural modeling using block definition diagrams allows the representation of these real-world entities as blocks, as well as the relationships among these blocks, in particular associations, whole/part (composition or aggregation) relationships, and generalization/specialization relationships. Composite relationships are used to show how a real-world system of interest is composed of blocks. In embedded systems, in which there are several physical devices such as sensors and actuators, block definition diagrams can help with modeling these real-world devices. Individual entities are categorized as input devices, output devices, timers and systems, and depicted on block diagrams using stereotypes.

As an example, consider the structural model of the problem domain for the Microwave Oven System, which is an embedded system and is depicted on the block definition diagram in Fig. 1. The Microwave Oven System is modeled as a composite component, which senses and controls several I/O devices through sensors and actuators respectively. The oven is composed of three input devices: a door sensor which senses when the door is opened and closed by the user, a weight sensor to weigh food, and a keypad for entering commands. There are two output devices: a heating element for cooking food and a display for displaying information and prompts to the user. There is also a timer component, namely the real-time clock.

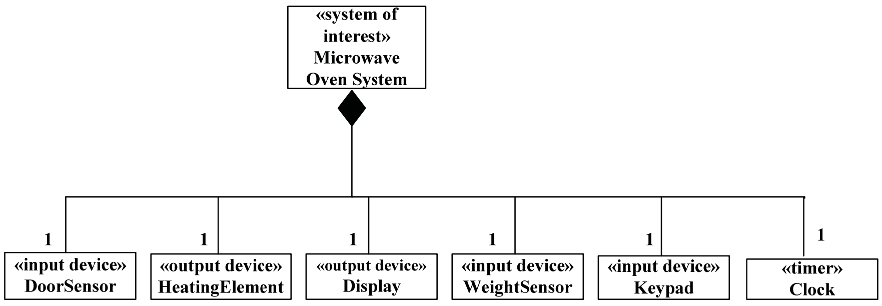

«system of interest» Microwave Oven System
1
«input device» DoorSensor
1
«output device» HeatingElement
1
«output device» Display
1
«input device» WeightSensor
1
«input device» Keypad
1
«timer» Clock

**Fig. 1 Block definition diagram for Microwave Oven System**

### 3.2 Structural Modeling of the System Context

It is very important to understand the scope of a computer system – in particular, what is to be included inside the system and what is to be excluded from the system. Context modeling explicitly identifies what is inside the system and what is outside. Context modeling can be done at the total system (hardware and software) level or at the software system (software only) level. A system context diagram explicitly shows the boundary between the system (hardware and software), which is treated as a black box, and the external environment. A software system context diagram explicitly shows the boundary between the software system, also treated as a black box, and the external environment, which now includes the hardware.

In developing the system context (which is depicted on a block definition diagram) it is necessary to consider the context of the total hardware/software system before considering the context of the software system. In considering the total hardware/software system, only users and external systems are outside the system, while hardware and software components are internal to the system. Thus, I/O devices are part of the hardware of the system and therefore appear inside the total system.

To differentiate between different kinds of external entities, stereotypes [4, 9] are used. For the system context diagram, an external entity could be an «external system», when the system to be developed interfaces to either a previously developed system or a system to be developed by a different organization, or an «external user», to model a user of the system.

As an example, consider the total hardware/software system for the Microwave System. The system context diagram (shown on the block definition diagram in Fig. 2) is modeled from the perspective of the system to be developed, which is the Microwave Oven System and is categorized as «system». External to the system is the external user (modeled as an actor, see below) who uses the oven to cook food.

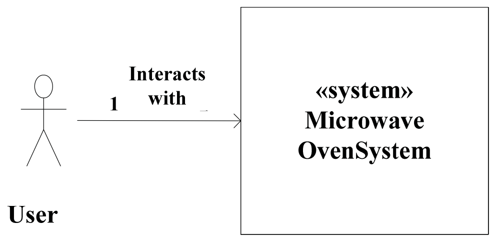


**Fig. 2 System context diagram for Microwave Oven System**

### 3.3 Use Case Modeling

In order to get an understanding of the system behavior, use case modeling [3, 4] is carried out. A use case describes a sequence of interactions between an actor (which is external to the system) and the system. In information systems, actors are usually humans. However, for embedded systems, actors could be also be external systems. In addition, a primary actor initiates the sequence of use case interactions. It is also possible to have one or more secondary actors that participate in the use case.

For the Microwave Oven System, the only actor is the user. In this simple example, there is one use case, Cook Food (see Fig. 3). The use case describes a main sequence in which the actor opens the door, places the food in the oven, closes the door, enters the cooking time, and presses the start button. The oven starts cooking the food and sets the timer. When the timer expires, the oven stops cooking the food. There are alternative to the main sequence, which the system has to be capable of handling, such as user opening the door before the food is cooked or pressing start before the time has been entered.

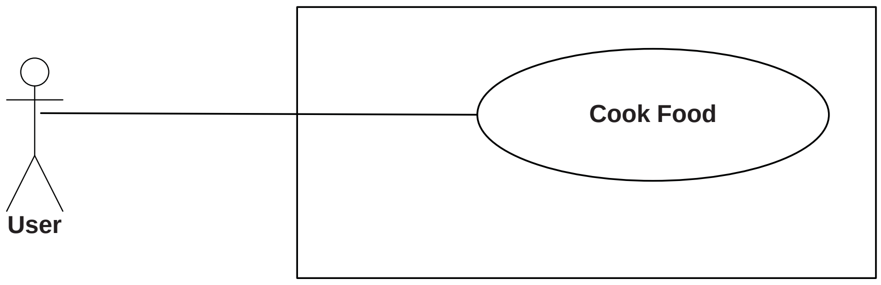


**Fig. 3 Use case diagram for Microwave Oven System**

### 3.4 Dynamic State Machine Modeling

Use case modeling provides an informal textual statement of requirements, which is usually sufficient for information systems but is unlikely to be precise enough for state-dependent embedded systems. State machines provide a more precise method for modeling such systems. A state machine can be developed by starting from the use case description and carefully considering all possible state-dependent scenarios. In particular, many events on the statechart correspond to inputs from the external environment, such as opening the door and placing the item in the oven, and many actions are outputs to the external environment, such as starting and stopping cooking.

The dynamic behavior of the microwave oven can be modeled as a state machine and depicted on a state machine diagram (also referred to as statechart [10]) as shown on Fig. 4, which depicts the states, events, and actions. The scenario described in the main sequence of the use case involves transitioning through the following states: Door Shut, Door Open, Door Open with Item (when item placed), Door Shut with Item, Ready to Cook (when cooking time entered), Cooking (when Start is pressed), Door Shut with Item (when timer expired), Door Open with Item, Door Open, and finally Door Shut. Many other transitions are possible corresponding to alternative scenarios.

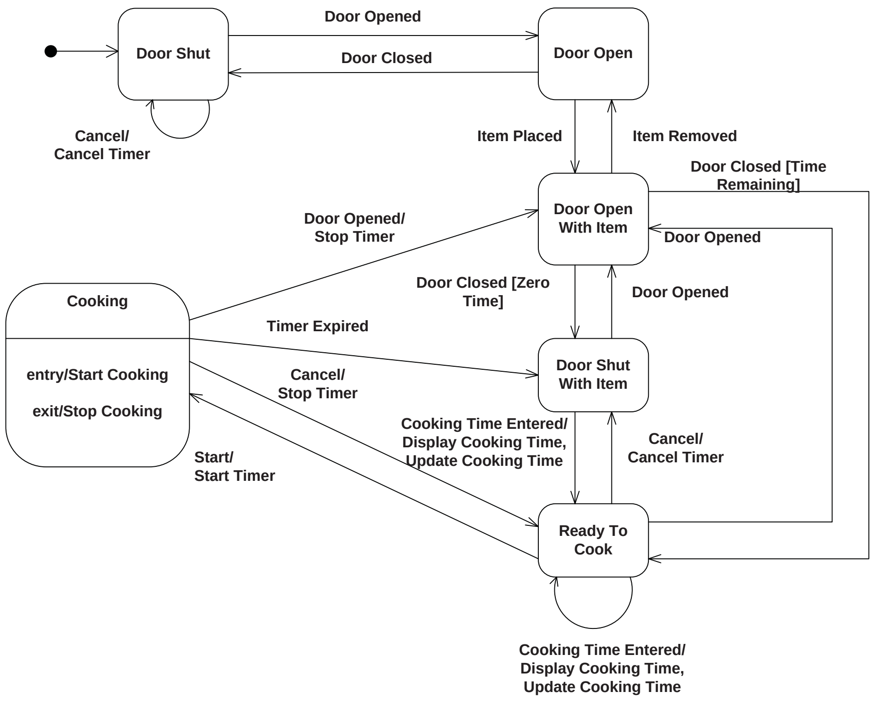


**Fig. 4 State machine diagram for Microwave Oven System**

## 4. Hardware/Software Boundary Modeling

Given the system requirements in terms of the structural model of the problem domain, the system context diagram, the use case model, and the state machine model, the system modeler can now start considering the decomposition of the system into hardware and software components. Hardware and software components are categorized using UML stereotypes.

### 4.1 Modeling System Decomposition into Hardware and Software Components

To determine the boundary between the hardware and software components, the modeler starts with the structural model of the problem domain (Section 3.1 and Figure 1) and then determines the decomposition into hardware and software components. The physical hardware components are explicitly modeled on the hardware/software block diagram while the software system is modeled as one composite component. In particular, the physical entities of the problem domain, as described in Section 3, are often input and/or output hardware devices that interface to the software system.

An example of a hardware/software system block diagram is given for the Microwave Oven System in Fig. 5. For this system, the devices originally identified in the structural model of the problem domain are analyzed further. The six part components of the Microwave Oven System identified in Figure 1 all have a hardware component to them, which are the hardware sensors and actuators that interface to the software system. There are three hardware input device components, Door Sensor, Weight Sensor, and Keypad (which all provide inputs to the Microwave Oven Software System), and two hardware output device components, Heating Element and Display (which receive outputs from the Microwave Oven Software System). There is also a real-time Clock hardware timer component, which signals the Microwave Oven Software System. The hardware components are categorized using UML stereotypes.

**4.2 Deployment Modeling**

The next step is to consider the physical deployment of the hardware and software components to hardware and software platforms. One possible configuration for the Microwave Oven System is depicted in the UML deployment diagram in Figure 6, in which the hardware and software components are deployed to different nodes physically connected by means of a high speed bus.

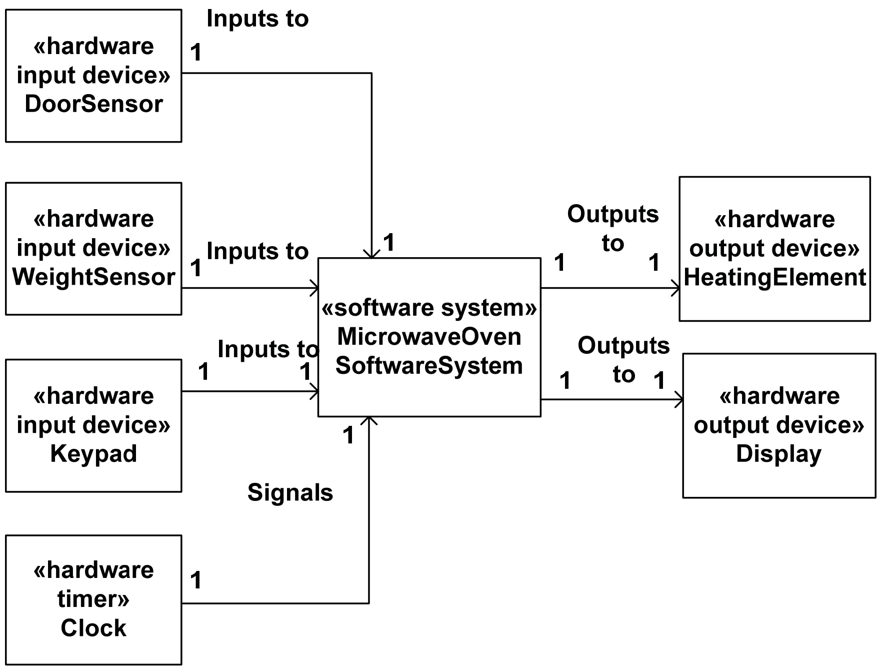

«hardware input device» DoorSensor
«hardware input device» WeightSensor
«hardware input device» Keypad
«hardware timer» Clock
«software system» MicrowaveOven SoftwareSystem
«hardware output device» HeatingElement
«hardware output device» Display
Inputs to
Inputs to
Inputs to
Signals
Outputs to
Outputs to
1

**Fig. 5 Hardware/Software block diagram for Microwave Oven System**

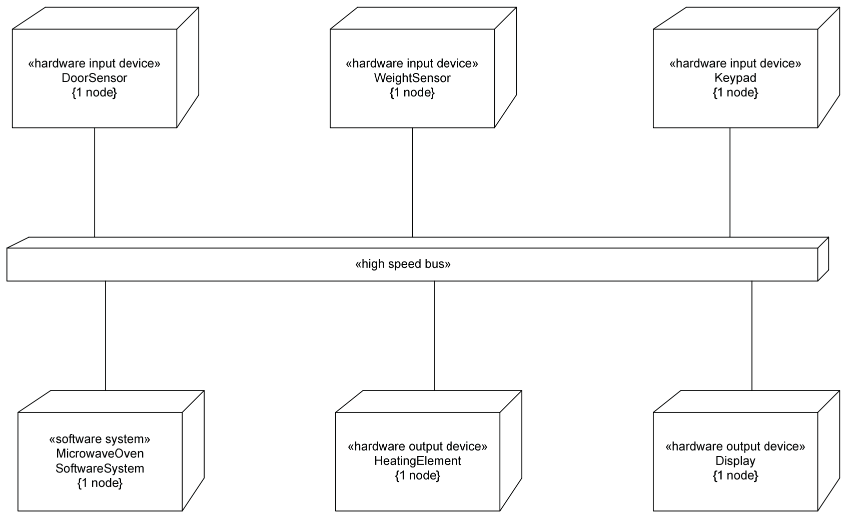


**Fig. 6 Deployment diagram for Microwave Oven System**

## 5. Software Modeling

The system context diagram depicts the systems and users that are external to the total system, which is modeled as one composite component. The hardware and software components are internal to the system and are therefore not depicted on the system context diagram. Together with the decomposition of the system into hardware and software components, this is the starting point for the software modeling.

### 5.1. Software System Context Modeling

The software system context model, which is used to model the boundary of the software system, is depicted on a class diagram and is determined by structural modeling of the external components that connect to the system. In particular, the physical hardware devices modeled on the hardware/software diagram are external to the software system.

The software system context diagram is modeled from the perspective of the software system to be developed, the Microwave Oven Software System, as shown in Fig. 7. From the software system point of view, the hardware sensors and actuators are external to the software system and interface to and communicate with the software system. Thus the hardware devices are external input and output devices, and an external timer as depicted in Figure 7, which is structurally similar to Fig. 5. However the categorization of the stereotypes is from the software system's point of view. Thus the hardware devices on Fig. 5 are categorized as external devices on Fig. 7.

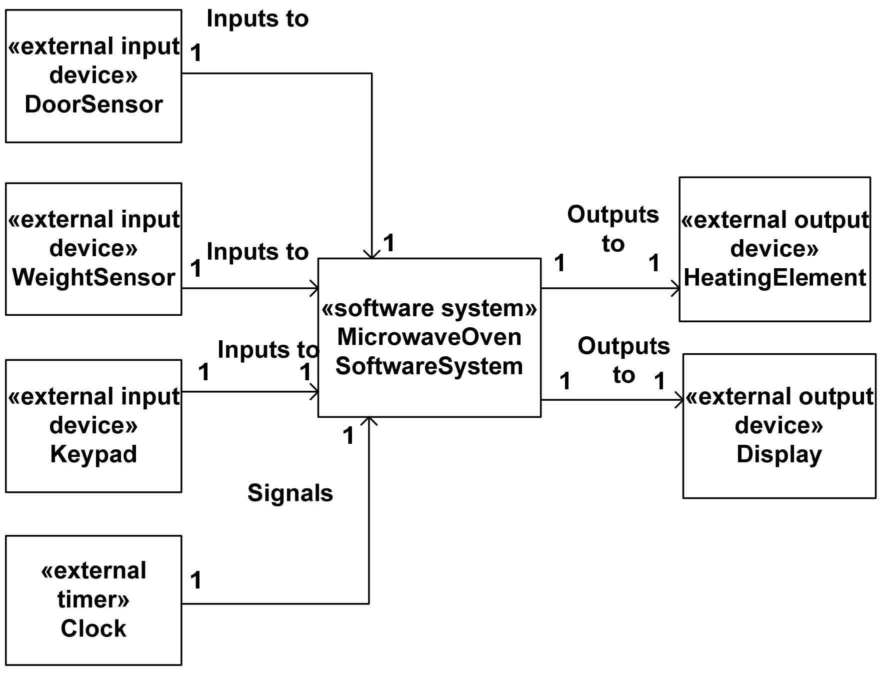

«external input device» DoorSensor
«external input device» WeightSensor
«external input device» Keypad
«external timer» Clock
«software system» MicrowaveOven SoftwareSystem
«external output device» HeatingElement
«external output device» Display
Inputs to
Inputs to
Inputs to
Signals
Outputs to
Outputs to
1

**Fig. 7 Software context diagram for Microwave Oven Software System**

### 5.2. Software Component Structuring

The next step is to determine the software components starting from the software system context model and working from the outside (hardware components) inwards to the software boundary components, which interface to and communicate with the hardware components. For every hardware component, there is a corresponding software boundary component, which is categorized using a UML stereotype. To receive input from an external input device, there needs to be a software input component. Each external output device component needs to receive output from a software output component. Each external hardware timer needs to signal a software timer component.

Software component structuring for the Microwave System is depicted on the class diagram in Fig. 8. Every external hardware component on the software system context diagram has a corresponding internal software component. Thus, there are three input software components, Door Sensor Interface, Weight Sensor Interface, and Keypad Interface, which receive inputs respectively from the Door Sensor, Weight Sensor, and Keypad external input devices. There are also two output software components, Heating Element Interface and Display Interface, which output to the Heating Element and Display external output devices, respectively. There are two additional components, a state-dependent control component, Microwave Oven Control, which executes the microwave oven state machine depicted in Figure 4, and an entity component, Oven Data, which contains data about the cooking time. In addition, there is one timer component, Microwave Timer, which receives periodic inputs from the hardware Clock.

### 5.3. Software Concurrent Task Design

A characteristic of real-time embedded systems is that of concurrent processing in which many activities occur simultaneously and the order of incoming events is frequently unpredictable [11]. Consequently, it is desirable for a real-time embedded system to be structured into concurrent tasks (also known as concurrent processes or threads).

During concurrent task design, the system is structured into concurrent tasks and the task interfaces are defined [4, 11]. As before, stereotypes are used to depict the different kinds of tasks. Each task is depicted with two stereotypes, the first is the role criterion, such as input or control. The second stereotype is used to depict the type of concurrency.

Thus, an active «I/O» component is concurrent and is categorized further using a second stereotype as one of the following: an «event driven» task, a «periodic» task, or a «demand» driven task. Stereotypes are also used to depict the kinds of devices to which the concurrent tasks interface. Thus, an «external input device» is further classified, depending on its characteristics, into an «event driven» external input device or a «periodic» external input device.

Figure 8 is the starting point for designing the concurrent tasks, which are depicted using the UML 2 notation of parallel lines on the left and right hand side of the object box, as depicted in Figure 9. The three input software components, Door Sensor Interface, Weight Sensor Interface, and Keypad Interface, are designed as event driven tasks, since they are awakened by interrupts from the respective input devices (see below). The two output software components, Heating Element Interface and Display Interface, are designed as demand tasks, since they are awakened by the arrival of messages from Microwave Control. Oven Timer is a periodic task since it is awakened by the arrival of timer events from the external clock. Microwave Oven Control is designed as a demand task, since it is awakened by messages from the input tasks or the periodic task.

The entity objects Oven Data and Display Prompts are passive objects and do not have a thread of control. Because the entity objects are passive, they cannot be deployed independently of other components. Furthermore the passive objects are composed into a composite component with the tasks that access the passive objects. Thus Microwave Control is a composite component with groups the two tasks, Microwave Oven Control and Oven Timer, which access Oven Data. Microwave

Display Interface is a composite component that groups Display Interface with Display Prompts.

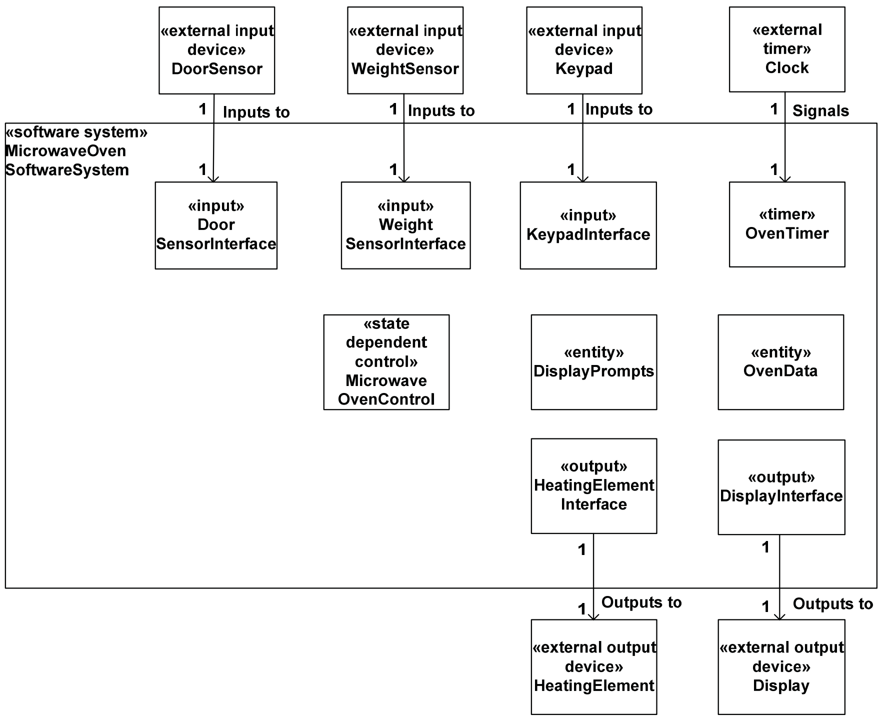

«external input device» DoorSensor
«external input device» WeightSensor
«external input device» Keypad
«external timer» Clock
1 Inputs to
1 Inputs to
1 Inputs to
1 Signals
«software system» MicrowaveOven SoftwareSystem
1
1
1
1
«input» Door SensorInterface
«input» Weight SensorInterface
«input» KeypadInterface
«timer» OvenTimer
«state dependent control» Microwave OvenControl
«entity» DisplayPrompts
«entity» OvenData
«output» HeatingElement Interface
«output» DisplayInterface
1
1
1 Outputs to
1 Outputs to
«external output device» HeatingElement
«external output device» Display

**Fig. 8 Software components for Microwave Oven Software System**

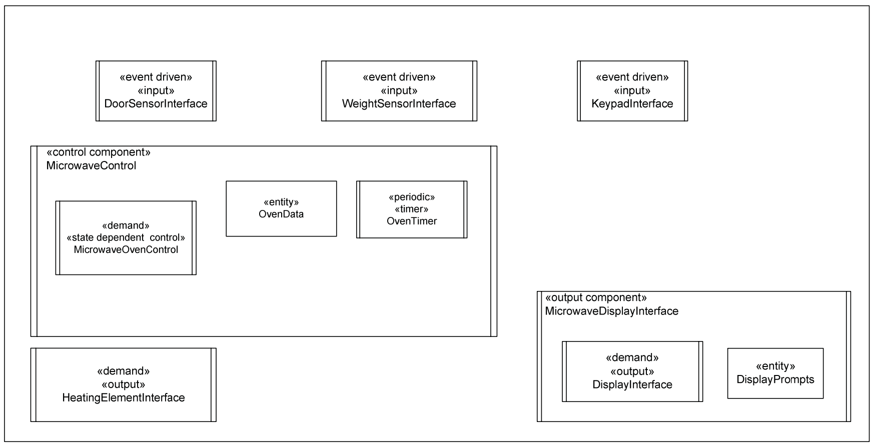

«event driven» «input» DoorSensorInterface
«event driven» «input» WeightSensorInterface
«event driven» «input» KeypadInterface
«control component» MicrowaveControl
«demand» «state dependent control» MicrowaveOvenControl
«entity» OvenData
«periodic» «timer» OvenTimer
«output component» MicrowaveDisplayInterface
«demand» «output» DisplayInterface
«entity» DisplayPrompts
«demand» «output» HeatingElementInterface

**Fig. 9 Concurrent components for Microwave Oven Software System**

### 5.4 Modeling and Design of Input/output Tasks

This section describes the concurrent modeling and design of the input/output tasks, since these tasks directly interface to and communicate with the hardware devices.

An event driven I/O task is needed when there is an event driven I/O device to which the system has to interface. The event driven I/O task is activated by an interrupt from the device, performs the I/O operation and then waits for the next interrupt. An example is given on the UML communication diagram in Fig. 10 in which the Door Sensor Interface event driven input task is awakened by an interrupt from the Door Sensor event driven external input device. This diagram uses the UML notation for active objects for the Door Sensor Interface task and the Microwave Control demand driven task.

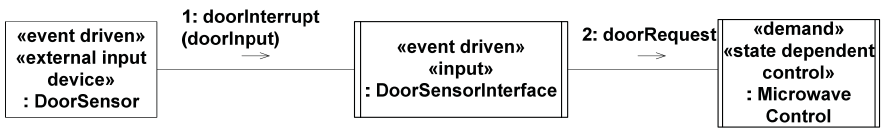


**Fig. 10 Event driven input task and demand driven control task for Microwave Oven Software System**

In the case of a passive device that does not generate interrupts, a periodic I/O task is developed to poll the device on a regular basis. The periodic I/O task is activated by a timer event, performs an I/O operation, and then waits for the next timer event. An example is given on the UML communication diagram in Fig. 11 in which the Temperature Sensor Interface periodic input task is awakened by a timer event from the Digital Clock, and polls the Temperature Sensor passive external input device.

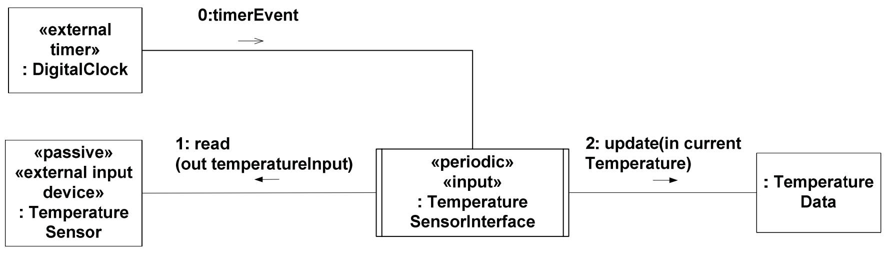


**Fig. 11 Periodic input task for Microwave Oven Software System**

## 6. Conclusions

This paper has described an approach for the integration of system modeling and software modeling, This approach is particularly useful for embedded systems, which are software intensive systems that consist of both hardware and software components. This paper has described a modeling solution to this problem with an approach that integrates system modeling using SysML with software modeling using UML for the development of embedded systems using both structural and behavioral modeling. In particular this paper has concentrated on the hardware/software boundary of a system with the decomposition into hardware and software components, and designing the interface between hardware and software components. The modeling approach described in this paper can also be extended to address the performance requirements of embedded systems [11] and to model system and software product lines [12].

## 7. References

1. Buede, D.M. *The Engineering Design of Systems: Methods and models*. New York: Wiley (2000)
2. Sage, A. P. and Armstrong, J. E., Jr., *An Introduction to Systems Engineering*, John Wiley & Sons (2000)
3. Booch**,** G. et al. *Object-Oriented Analysis and Design with Applications*, 3rd ed. Boston: Addison-Wesley (2007)
4. H. Gomaa, "Software Modeling and Design: UML, Use Cases, Patterns & Software Architectures", New York: Cambridge University Press (2011)
5. M**.** Blaha and J. Rumbaugh**,** *Object-Oriented Modeling and Design with UML*. Upper Saddle River, NJ: Prentice Hall (2005)
6. Douglass, B. P. *Real Time UML: Advances in the UML for Real-Time Systems*, 3rd ed. Boston: Addison-Wesley (2004)
7. S. Friedenthal, A. Moore, and R. Steiner, A Practical Guide to SysML: The Systems Modeling Language, Morgan Kaufmann (2009)
8. Booch**,** G.**,** J**.** Rumbaugh**,** and I**.** Jacobson. *The Unified Modeling Language User Guide*, 2nd ed. Boston: Addison-Wesley (2005)
9. Rumbaugh**,** J.**,** G**.** Booch**,** and I**.** Jacobson**.** *The Unified Modeling Language Reference Manual*, 2nd ed. Boston: Addison-Wesley (2005)
10. Harel, D. and E. Gary, "Executable Object Modeling with Statecharts", Proc. 18th International Conference on Software Engineering, Berlin (1996)
11. H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Boston: Addison Wesley, (2000)
12. Gomaa, H., Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. Boston: Addison-Wesley (2005)

# A Multidisciplinary Design Methodology for Cyber-physical Systems

Frank Slomka, Steffen Kollmann, Steffen Moser and Kilian Kempf *

Institute of Embedded Systems / Real-Time Systems
Ulm University
{frank.slomka, steffen.kollmann, steffen.moser, kilian.kempf}@uni-ulm.de

**Abstract.** Designing cyber-physical systems is a challenge originating from the multidisciplinary and mixed-signal requirements. In order to handle this challenge, many design languages have been developed, but none is able to connect different application domains adequately. This paper proposes a new system based view for cyber-physical system design which can be easily adapted by MARTE or SysML, as it uses a model based design technique. Instead of defining another UML profile, we present an intuitive idea for the development of cyber-physical systems by refinement and introduce new abstraction layers that help to describe operating system and mixed-signal issues. Using new abstraction layers, it is now possible to support all views of the platform based design by using one consistent language. The approach explicitly distinguishes between the physical system and the computational system. The benefit of this new approach is presented in a case study where a cyber-physical system is designed.

## 1 Introduction

The design of cyber-physical systems [14] – consisting of software as well as digital and analog hardware – is still a great challenge that is caused by the increasing complexity and the multidisciplinary requirements which are typical for mixed-signal applications. One issue is to connect different application domains of the system in a whole design process. To cope with this, many different design languages have been developed.

Model-based design with the Unified Modeling Language (UML) [20] is a common way to model software systems. During the last years it has been adapted to embedded systems. For this, UML has been extended by the profiles Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [18] and OMG Systems Modeling Language ((OMG) SysML) [19]. A lot of different functional and extra-functional[1] diagrams and models are defined in both modeling languages. UML diagrams in MARTE are defined for embedded system design and support a lot of different views. However, a closer look at the specification of MARTE shows that the authors are software oriented. System aspects like the description of physical behavior with differential equations are

* This work was supported by the German Research Foundation.

[1] Extra-functional is similar to non-functional. But in our opinion, each requirement is needed for the functionality of a function. Therefore, requirements like time are considered as extra-functional.

marginal. However, for both, MARTE and SysML, the platform-based design is not considered well. Certainly it is possible to define hardware architectures as well as the binding of computational elements to processing elements, but the language does not support hierarchical bindings and does not support operating system issues and physical design aspects needed in mixed-signal design. Therefore the system view in the Y-diagram of platform-based design [6] is poorly implemented in MARTE and UML. SysML does not support graphical representations for different types of physical and logical flows [19]. In SysML, the attribute of a flow is described by using flow properties. However, to support engineering of cyber physical systems, a clear distinction between different classes of types is needed. Such an approach is presented in this paper.

Another possibility to specify cyber-physical systems is the use of process models like the ANSI/ISA-5.1-1984 (R1992) standard [11] to describe measuring and control devices. For example, it allows the modeling of devices used by control room operators. Such a process model is able to describe different aspects of the physical environment and the connection, but the design of the computer architecture is not covered in detail.

It is inherent to the design of cyber-physical systems that the design process has to cover different application domains like automotive, avionic, industrial control, mobile communication, etc. Each domain has different views on technical and physical details. As a consequence to cyber-physical systems design, the design methodology has to consider all these different aspects. To cope with this problem, domain specific languages have been developed to cover the design challenges of specific systems. For example, the tool PREEvision of the company aquintos [3] can be used as design entry in the automotive domain. Unfortunately, other domains are not covered.

To handle the problems discussed above, we introduce a new approach for system modeling. We supply a new idea to support cyber-physical system design by introducing new symbols. These symbols are independent from UML, MARTE, or SysML but can be easily adapted to them. The goal of the methodology is to give the designer the opportunity to refine a system during design. The approach extends the object-oriented philosophy of designing software systems to multidisciplinary, multi-technology hardware/software systems. Therefore the methodology supports application design as well as platform design in a single view. This approach is only suitable if the methodology supports the refinement process and a refinement history. It should be possible to move in both directions of the refinement process, an approach that could be compared to the possibilities of version control tools.

Using such a new description language, it is necessary to include it into the design flow of cyber-physical systems. This is essential and often not sufficiently considered when new languages are designed. For example, in [23] a generic design flow for embedded systems is presented, but the granularity is not sufficient for our idea. A design flow for the automotive domain is presented in [26] and is confirmed by a real example in [13]. But in that contribution, only the automotive domain is covered. No adequate design flows exist that are suitable for cyber-physical systems and therefore we introduce an appropriate design flow in Sect. 2.

The remainder of the paper is as follows: The different abstraction layers are presented in Sect. 3. In Sect. 4, the new system model is introduced. After this, an exhaustive case study is given (Sect. 5), followed by a conclusion.

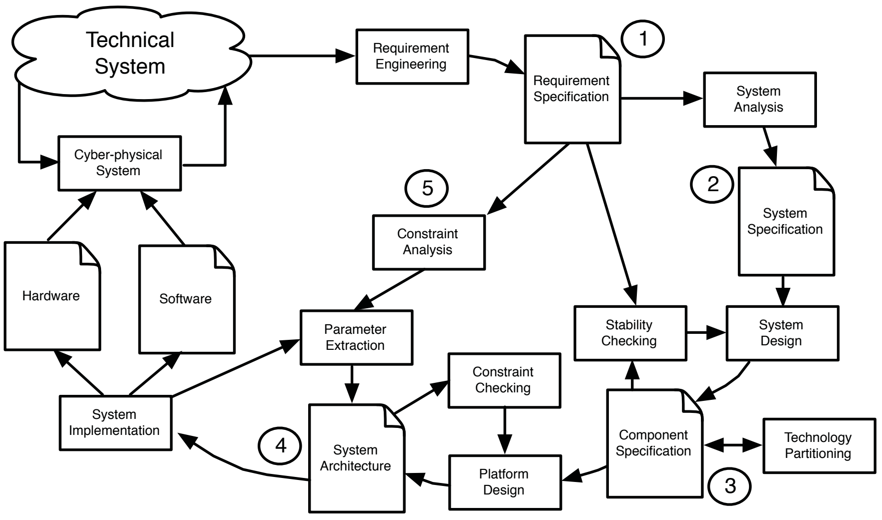


**Fig. 1.** Generic design flow of cyber-physical systems

## 2 Design Flow

Figure 1 gives an overview of the proposed design methodology which adapts object-oriented design methodologies as given in [12] to a multi-technology component approach.

In the first step, well known from software system techniques, a requirement specification is conducted ①. After finding the requirements, the system's functionality has to be worked out. For this, a system analysis based on the requirements is performed. It will identify which subsystems are needed and how the information, material, or energy flows between these subsystems. Note that this step is in most cases not a software or information technology problem. It is an engineering problem, and the information technology does not play an important role at this stage. The result is a small model-based system specification which considers all aspects of the whole system.

Based on the resulting system specification ②, the system design starts. At this stage, the previously specified subsystems have to be refined and the underlying technology has to be chosen. Both parts, system design and component specification ③, are done on the application view [8]. The system and component design follow the well known steps of object-oriented methodologies, called system analysis and system design, where the design flow bases on a refinement technique. This means an engineer can first verify the system's stability independently from the computer platform which is designed later. This is an important aspect of controller dominated applications. The next step is to perform a technology partitioning. The subsystem has to be partitioned into different technology domains. This means the designer has to assign which part of the system has to be implemented in mechanical, electrical, or computational hardware and which technologies are used to realize the implementation of the hardware. Be aware that at this point in the design flow no decision concerning software is made. At this stage we distinguish only between mechanical, analog and digital.

The step after the technology partitioning is the platform design, where components, tasks, or controllers are mapped to allocated computational resources. The platform design contains several design steps like the allocation of processing elements, communication elements, memory elements, scheduling spaces or address domains, and the binding of tasks to scheduling spaces, tasks and buffers to address spaces, scheduling spaces to processing elements, or address spaces to memory components. The result is the system architecture ④.

The next step is the constraint analysis ⑤, which is well known as hardware/software co-design or system synthesis. Further details can be found in [25] and [7]. During this design step, a constraint checker verifies the chosen computer architecture and the schedule of the chosen binding. It delivers platform parameters for the component model.

After this, the system's stability considering the influence of the platform on the application and the technical system can be verified. Based on a design space exploration step to find the cost optimal platform that allows a robust operation of the whole system, the design of the hard- and software starts. After implementing hardware and software, model parameters are extracted from the implementation specifications and the model of the cyber-physical system is checked against the specified constraints.

Considering the whole process, the system design usually starts with the platform design immediately after the system specification. This means the global view on the system design is still missing. In the following sections we will close this gap with a new system view that allows to consider multidisciplinary design criteria during the design process of cyber-physical systems. Thereby a refinement process is supported to enable stability corrections at a high level.

## 3 Refinement

After introducing the design flow of cyber-physical systems, we now give a closer look at the refinement process which is included. We distinguish between the hardware, software, and system refinement. It will be discussed that the classic abstraction layers of hardware and software are not sufficient for complex cyber-physical systems.

### 3.1 Hardware Abstractions

From hardware design, six different layers of abstractions are known. These layers are well established in the semiconductor industry and each is supported by different models of computation and verification tools. The classification is given in [8] and is divided into system level, behavioral level, register transfer level, gate level, transistor level, and layout level. As hardware design is well understood, we will not deal with these levels anymore. In the design flow given in Fig. 1 we find these parts downwards from the platform design.

### 3.2 Software Abstractions

Software development differs from hardware development. However, a closer look shows that software development can also be separated into different abstraction

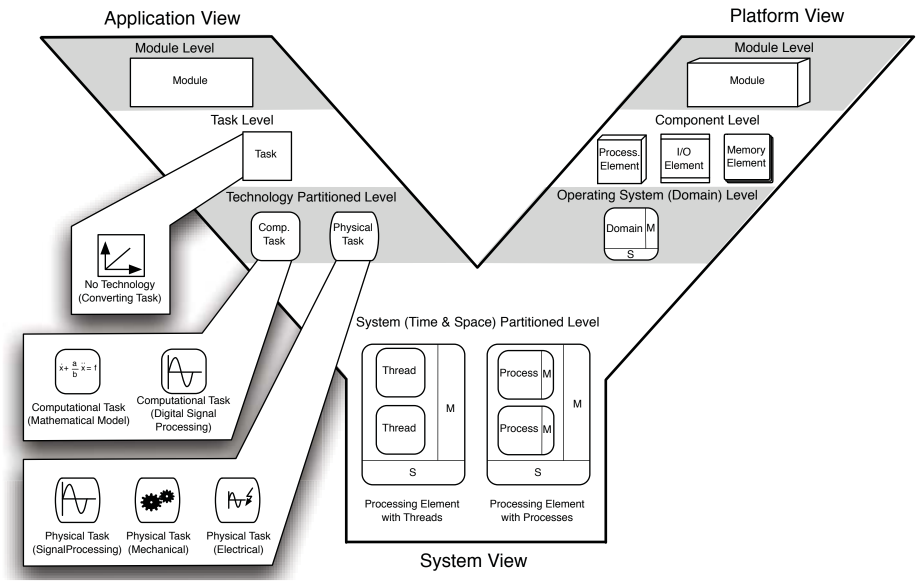

Application View
Platform View
Module Level
Module
Task Level
Task
Technology Partitioned Level
Comp. Task
Physical Task
No Technology (Converting Task)
Computational Task (Mathematical Model)
Computational Task (Digital Signal Processing)
Physical Task (SignalProcessing)
Physical Task (Mechanical)
Physical Task (Electrical)
Module Level
Module
Component Level
Process. Element
I/O Element
Memory Element
Operating System (Domain) Level
Domain M S
System (Time & Space) Partitioned Level
Thread
Thread
M
S
Process M
Process M
M
S
Processing Element with Threads
Processing Element with Processes
System View

**Fig. 2.** Abstraction levels in the design flow

layers. This classification is adapted from [24] and is divided into architecture level, component level, algorithmic level and machine code level. The abstraction starts at the platform design as well as in hardware design, because the platform is needed to start the software design. The result is that the previous system design as given in Fig. 1 is not covered by these abstraction layers. Therefore we define a new approach to system abstraction on higher levels.

## 3.3 System Abstractions

Since the introduction of the system-on-a-chip paradigm, the conventional model does not apply anymore. The common approach ends below the system level, but systems have to be consistently explained in the whole. Therefore a new paradigm has been developed to support system synthesis. This *platform based design* is shown in Fig. 2 by a Y-structure. It is distinguished between the different views Application, Platform and System, as further discussed in [22]. This means the development of the application's functionality is separated from the development of the hardware platform. Hence, this approach takes into account that in many projects, hardware platforms are used for different products. The system axis describes the mapping or binding of application functions to the hardware components. Although the application and the platform view is covered by UML and AUTOSAR [1], the system view is not supported well by these techniques.

We introduce the abstractions top-down and start with the module level in the application view. Normally, the mission level is the starting point as introduced in [4], because many cyber-physical systems are part of larger distributed systems. For example an autonomous underwater vehicle contains many different subsystems which interact: the sonar system, the navigation, the actuator of the vehicle, and the control unit. The interaction of all these components is considered on the mission level. For this paper, the mission level is not considered any further.

**Application View** The application view is supported in three abstraction levels: We distinguish between the module and the task level. The task level itself is separated into two different levels: The first one is a level at which a designer only concentrates on the functionality of the system. This models the middleware abstraction of a system which is comparable to the AUTOSAR [1] standard for the automotive domain whereby our middleware is not domain dependable. It is done by the composition of the several tasks describing the behavior of specific system parts. Each behavior is encapsulated by a task and tasks may communicate with each other. Note that in this context a task does not mean an operating system task or process. In other words, only entities of functions are composed. In the second task level, the technology partitioned level, design decisions for the mapping of tasks to technologies are described. At this stage of the development process the designer documents the choice between digital hardware and analog electronics or mechanical elements. The computational tasks are mapped to the platform in a later stage described in detail in the step System View.

**Platform View** The platform view considers only the hardware for the computational tasks. The process starts with the module level to describe the approximated platform. The refinement step Component Level defines the system platform in more detail. At this level the processing elements (like CPU or buses), I/O elements, and memory elements are specified. The resulting architecture from the component level can then be refined at the operating system level. At this stage the different components are divided into domains, describing the scheduling behavior and memory architecture of the platform.

Note that this is an abstraction to support space and time partitioned platforms for the applications. For the application it is not important how the components are connected, it is important which memory can be used and when it gets resource time. Therefore this level supports an abstraction for operating systems.

**System View** The system view unites the application and platform view. At this level application tasks are mapped or bound to the elements of the platform. Each computational task is assigned to a platform domain describing its scheduling and memory architecture. Based on the mapping it is possible to classify if a task is implemented as a thread sharing memory with other threads or as a process having its own memory.

In MARTE, allocation means the mapping of application to a platform [5]. In system synthesis, allocation means to choose a component, while binding is the term to describe mappings of tasks to allocated components. As described in [5], MARTE distinguishes between structural and behavioral bindings. In this paper we use the system synthesis terms instead, as the aim of the methodology is to synthesize embedded systems. As shown in [5], the binding mechanism in MARTE does not consider that a task needs both a computational resource as well as memory resources. If this is modeled in a naive way, for each part – computational binding and memory binding – different allocation arrows are needed. In complex architectures this is not an elegant way because of the rising complexity. This approach is given by MARTE (see pages 132 ff. [18]).

In this paper we introduce domains. Domains are an abstract model of resources and can be modeled using the service construct in MARTE. However, services are not the concept needed by synthesis, because services hide architectural aspects. Therefore domains are a special kind of modeling element, as they allow an abstract formulation of architectures. Not in the detailed way like component diagrams, but more in an abstract way. Using the concept of hierarchical composition of domains, it is possible to describe complex hardware architectures easily. Such a model is needed because a complex binding relationship as proposed by MARTE is very hard to use if an automatic design space exploration has to be designed and implemented because there exist a lot of dependency rules between the different views. A comparable approach is presented in [15]. However, the activity threads discussed in that paper are mappings of the detailed architecture to the application. This is equivalent to modeling the binding with application arrows and does not reduce the complexity. Additionally, our approach explicitly supports the design refinement of cyber-physical systems. In MARTE or SysML this is only supported by using attributes.

In UML, MARTE, or SysML it is only possible to describe the application or platform view. Binding and mapping is only supported in a graph based approach like in [25]. Memory is not considered in it. The whole system view is also missing in all cases. This gap is closed in this paper by the newly introduced system view.

## 4 System Description Language

In this section we introduce our new system description language as a design entry for cyber-physical systems. As discussed in the last section, this view allows to support the refinement process in the design flow as well as the connection of different technology domains into one model. In contrast to other models, the design of the computer system is still possible, because the methodology can be easily adapted to UML/SysML and other techniques. Models given in a graphical specification language have to be intuitively understandable. As mentioned, this is one reason for the legitimacy of domain specific languages. Our intention is to consider the domain of cyber-physical systems in general. Limitations of the approach as described in this paper are a missing binding from constraints to clocks. The analog part is just drafted in the paper to separate it from the digital part. Therefore the presented methodology does not allow a compositional design flow on the analog parts of the cyber-physical system.

### 4.1 Application View

To support a multi-technology model, different symbol types are defined for the application view, which are depicted in Fig. 2. First of all it is distinguished between modules and tasks. Modules are blocks containing other design entities such as modules and tasks. They are used to support a hierarchical design methodology. A module may contain different tasks. However it is also possible that a module contains mathematical models, functions, or formulas. There are different types of modules specified to support intuitive and readable system models: modules keeping mathematical models, modules keeping electronic

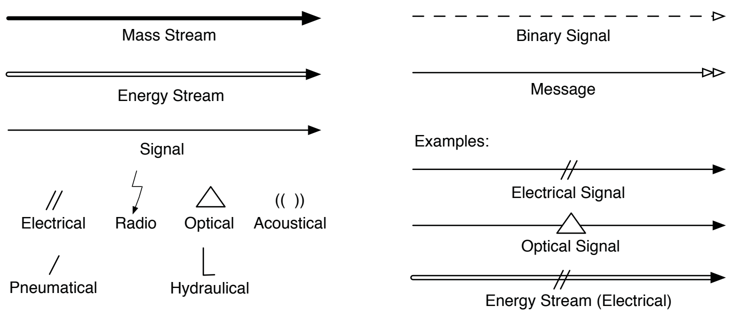


**Fig. 3.** Symbols of communication

systems, modules keeping electrical models, and modules keeping mechanical models. These types are supported in all abstraction levels as shown in Fig. 2. So a task can be a converting task where the technology is not specified. But it is also possible to have tasks with a technology binding and a type.

## 4.2 Platform View

The platform view also starts with modules to support a hierarchical design methodology. The modules can consist of several modules or components. Components are hardware elements such as memory, processors or processing elements and interface elements. Therefore components are model elements of a hardware view. The components can then be refined by domains. A domain in the platform view describes on the one hand the access to the resource through a scheduling strategy and on the other hand the memory architecture in which the domain is embedded.

## 4.3 Communication

To connect the different tasks in the application view, different types of communication mechanisms exist. They are depicted in Fig. 3. As well known from hardware description languages, the communication links are called signals and messages. The following classes of signals are supported: 1. Mass stream: Transport of mass. 2. Energy stream: Transport of energy. 3. Signal: Transport of information. 4. Binary signal: Computational signals as known from hardware description languages with the binary states, like true, false, undefined and high-impedance. 5. Messages are only supported in computer systems. It is also possible to define message types with complex payload data unit structures, like integer or string messages. Additional symbols are added to the communication links to characterize the link: electrical, radio, optical, acoustical, pneumatical and hydraulical types are supported. This is illustrated by three examples in Fig. 3.

Signals of different semantics can only be connected by using transformers. An example is given later in the paper: a transducer transforms an electrical signal to an acoustical/mechanical signal. Therefore design tools have to check the type of a language construct in order to separate different views and to avoid connecting different incompatible components.

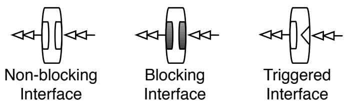


Fig. 4. Symbols of interfaces

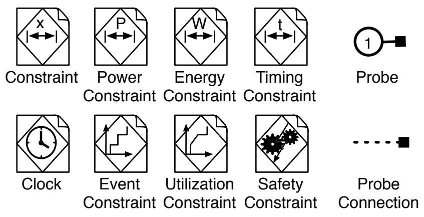


Fig. 5. Symbols of requirements

### 4.4 Interfaces

Figure 4 gives an overview of different graphical symbols that model interfaces to computational tasks. An interface is generally specified by a rectangle with round sides. It is distinguished between different types of interfaces that have been partly inspired by the abstract communication channels presented in [9]. There are blocking, non-blocking, and triggered interfaces. Blocking means that the sender is blocking until the data is received by the receiver. Non-blocking means that the data is stored until the receiver is ready to receive the data. Triggered means that the receiver task is triggered to receive the data. Other possibilities for interfaces between computational tasks can be found for instance in [10].

### 4.5 Constraints

To specify extra-functional constraints, graphical elements are defined. Such elements are a diamond in a typical comment symbol defining the type of the constraint as shown in Fig. 5. We distinguish between timing constraints, area or cost constraints, power and energy constraints, and safety constraints. Due to the compositional approach of the diagram types it is very easy to define additional constraints. Note that requirements are always specified between probes.

Assume that in this paper the time model presented in [17] is used instead of the models given in MARTE. This is done because MARTE does not give a formal semantic to its model and the MARTE model is just a syntactic extension to the time model of [17].

## 5 Case Study

In this section, a design example is considered. The sonar system is part of a larger project, an autonomous under water vehicle (AUV). The major task of the sonar system is to detect objects in the water that will be used for navigation and maneuvering of the robot. The sonar system sends acoustic waves into the water and, if there is any object, receives the acoustic reverberation of that object. Such a system is a typical example of mixed technology domains. It consists of an electromechanical part to generate acoustic signals, an analog electronic part to drive the electromechanical sound generator and to receive the reverberation, and a digital electronic part with hard- and software for signal processing and target detection. We divide the design process into three parts. The first one is

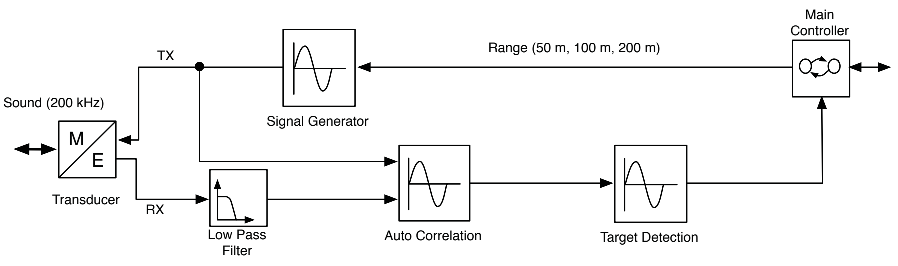


**Fig. 6.** Analysis components

the system design, starting with the requirements analysis. In the second step the platform design is conducted. Finally, the system is implemented with well known procedures for hardware and software design.

## 5.1 System Design

We start with the system design. The focus here is to perform the requirements engineering and to conduct the system analysis. Based on this, the main components are specified and the technology partitioning takes place.

**Requirement Analysis** The sonar has the ability to detect underwater objects (targets) by using sound signals of in this case 200 kHz. It sends an acoustic signal and then waits for echos from the targets. In order to accomplish that, the sonar has to measure the time from the emission of the signal up to the reception of the echos generated by several targets. The system should be able to detect objects in the range from 2 m up to 200 m. To support a high resolution and a fast detection of narrow targets, the range has to be chosen from three ranges: 2–50 m, 2–100 m, and 2–200 m. These ranges are available in different modes of the sonar.

**System Analysis** The second step in the design process is the application analysis, as shown in Fig. 6. In this step, the main functionality of the system is specified. The design of an actor oriented system starts with its design entities. In the case of the sonar system, the following analysis objects are defined: A signal generator that forms the sending signal, a transducer that transforms the electrical signal into an acoustic signal, and a receiver that filters the received signals. This step is mandatory because the signal strength of the received signal is very low compared to the strength of the sending signal. After receiving the echos of detected objects, an auto correlation function detects the echos of the sending pulse (see Fig. 6). This means that the pulse form of the received signal must be the same as the pulse form of the sending signal. After filtering, only signal echos remain. A target detection then analyzes the echos and determines the target's destination. The whole system is controlled by a main controller. In this phase, the main controller, the low pass filter, and the transducer are modeled as tasks, while the signal generator, the auto correlation, and the target detection are drawn as blocks. These blocks will be divided into subtasks during the following design steps.

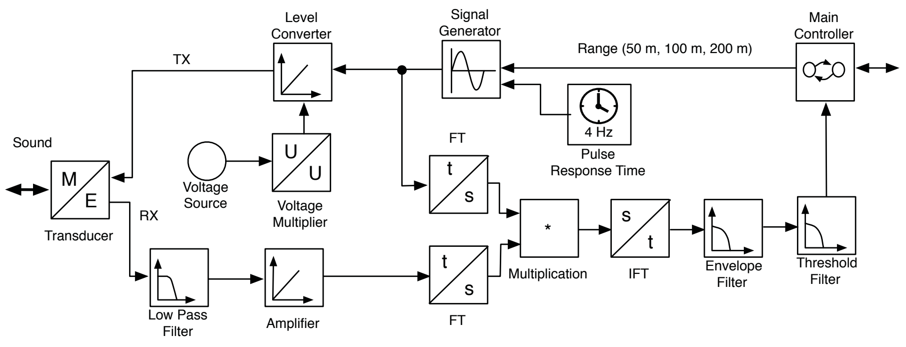


**Fig. 7.** Design components

**System Design** In the next step, the application design has to be refined. How is the auto correlation and the target detection implemented, and how does the signal generator work? The resulting application design is shown in Fig. 7.

*Signal Generator:* The sonar signal will be generated by a signal generator which forms digital pulses. A level converter transforms these pulses to the voltage level needed by the acoustic transducer, an X-cut crystal which transforms electrical signals into acoustic waves. The transceiver sends pulses of 200 kHz as given in the requirements. To prepare the signal detection, the sending signal generated by the signal generator is transformed by a Fourier transformation.

*Auto Correlation:* After amplifying the signal, the echo detection is performed by an autocorrelation function which can be implemented with a signal convolution. The autocorrelation identifies the sent pulse in the received audio spectrum. This step is necessary to be able to detect the received signal against the background noise of the sea. A convolution contains an integration of the received signal. However, it is known from signal and system theory that such an operation in the time domain can be transformed into a multiplication in the frequency domain. This transformation is performed by the Fourier transformation. The multiplication then may perform the convolution or the correlation function.

*Target Detection:* After transforming the signal back into the time domain, the targets are detected by an envelope filter and a threshold filter. The envelope filter reduces the information of the signal to just the interesting envelope while the threshold filter detects the received pulses and their timing. The timing of the received pulses is then sent to the main controller which calculates the range of the detected objects. The sonar data can then be sent to other system components like the navigation module of the robot.

**Technology Domain Binding** The first step in system refinement is the domain binding. The system architect has to decide which parts of the system are implemented in which technology. As seen in Fig. 8, the transducer and the level converter are implemented in the physical electronic domain. The system architect decides to implement the signal processing in digital electronics because the maintainability of digital systems is better and the design of digital electronic is easier than the design of analog components. However, to support

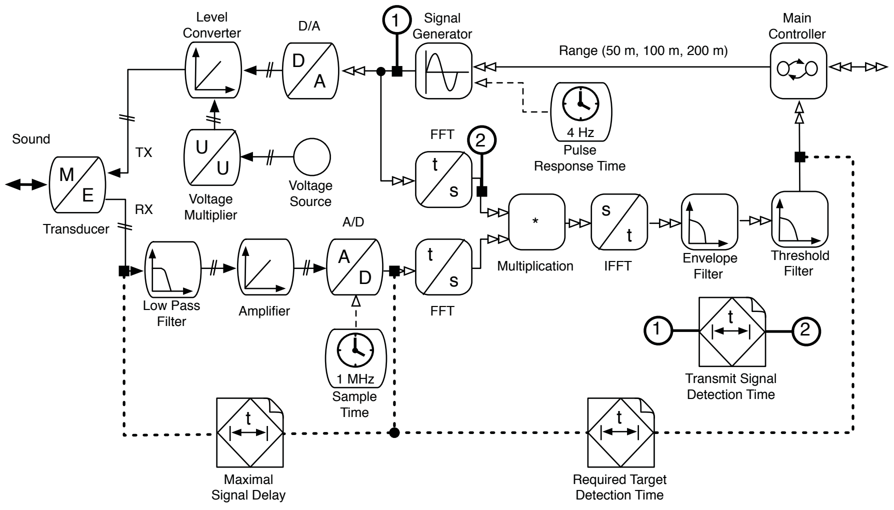


**Fig. 8.** Technology binding with added probes and timing constraints

this decision, a new component must be added to the signal flow. An analog digital converter is needed to discretize the received analog signal. As seen in Fig. 8, the Fourier transformation, the signal multiplication the signal generator, the inverse Fourier transformation, the envelope and threshold filter, and the main controller are now refined into digital processes. After binding tasks to a technology domain, electrical signals are refined into binary signals or messages.

**Timing Requirements** Fig. 8 also shows the timing requirements, which are added in this step. To sample 200 kHz signals, a sampling rate of twice that frequency is needed. However, the sampling rate of the analog-digital conversation is chosen to be 1 MHz, because the low pass filter is not perfect and the signal contains parts with frequencies above 200 kHz. So additionally, three timing specifications are added: the sampling rate of the analog-digital conversion, the required time for the target detection, and the overall computation time of the system. The time required for target detection was calculated in the following way: As given by the system specification, the sonar has three detection ranges. A short range up to 50 m, a mid range up to 100 m, and a long range up to 200 m. According to the average signal speed in water, a signal from an object in a distance of 50 m is received after 71 ms, so after sending a pulse, the system has to wait for this time before sending the next pulse. Because the 50 m range is the fastest system mode, a target detection time of 71 ms is needed. From these considerations, the maximal pulse response time may be calculated. To support the auto correlation, an additional timing requirement is needed, the transmission signal detection time. The Fourier-transformed sending signal has to be stored before the first echos are received. As stated in the AUV specification, the minimal detection range is 2 m, which leads to a transmission signal detection time of 3 ms.

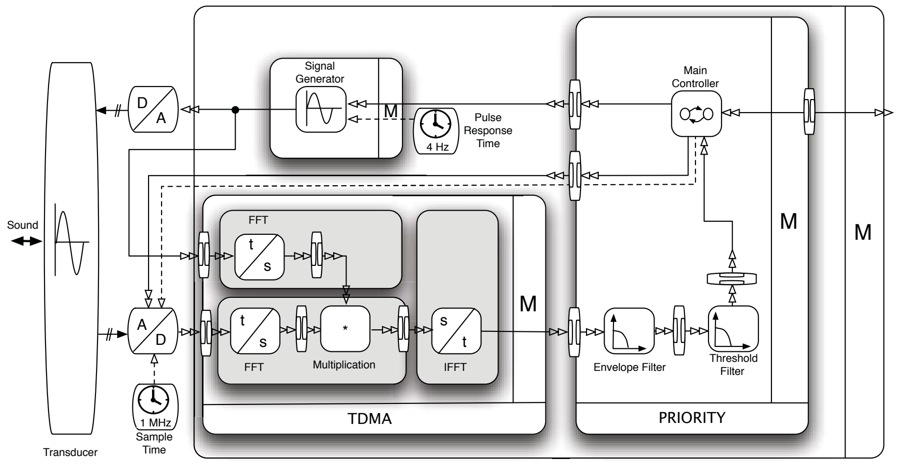


**Fig. 9.** Architecture of the sonar

## 5.2 Platform Design

After the definition of the application, the platform has to be designed. In this step, the hardware/software architecture of the system is devised. This means the required circuit techniques are selected in the analog domain, while in the digital domain, hardware technologies, hardware architectures, processing components, and software architectures are designed.

The first step is to specify buffers in the form of non-blocking interfaces. Each process has to be able to communicate with other processes through messages. To model the message communication, buffers are added to the model. Their size can not be specified at this first step because it depends on the scheduling and binding of the processes. After adding the buffers, the scheduling and memory spaces are selected. This step is an optimization process and will be performed with the help of a design space exploration.

**Binding and Allocation of Computation** A typical example is shown in Fig. 9. In this example, a hardware component which implements the signal generation, an instruction set processor and a coprocessor are allocated. The coprocessor implements the fast Fourier transformation (FFT) and a subsequent multiplication. The FFT coprocessor can also be used to execute the inverse Fourier transformation. This is the reason why the IFFT is bound to its own scheduling domain. All FFT operations are scheduled to the hardware coprocessor FFT that also performs the multiplication and on which a TDMA (Time Division Multiple Access) scheduling is employed. The main controller and the two digital filters are mapped onto the instruction set processor and are scheduled by a fixed-priority system. The hardware/software model is analyzed in the next step. The architecture presented in Fig. 9 can be directly used as a specification for Symta/S [21], the real-time calculus [27], or the event-spectral calculus [2] by adding the required model parameters like the worst- and best-case execution times. The event models necessary for the analysis can be derived directly from the specification as well as the timing requirements.

**Binding and Allocation of Memory** In embedded systems, the platform often consists of more than one processor. In our example, two application specific hardware processors and one instruction set processor are used. The data spaces of the tasks and the message buffers for the interprocessor communication has to be bound to the system's address space. One solution could be the allocation of one memory element and the binding of all buffers and process data areas into one address space. However, there are several possible solutions. The FFT processor may have its own memory and the buffers to and from the FFT processor are bound to that memory. In order to perform a communication to the filters implemented on the instruction set processor, the buffer between this task then has to be implemented on a shared memory.

### 5.3 Implementation

The last step is to implement the system based on the embedded architecture. From here, well known established methodologies can be used to implement the software and hardware components as introduced in Sect. 3. For example, the system can be defined in Matlab/Simulink [16], which will provide an appropriate model for the verification. Based on code generation toolboxes it is possible to generate the hardware description language and software code for the implementation. Using automated back end processes leads to the desired implementation.

## 6 Conclusion

In this paper we have presented a new methodology for the description of cyber-physical systems. We have introduced a new way to model such systems in the whole. The main focus was the definition of a new entry for the design process, covering multidisciplinary design constraints. In contrast to other models, the presented approach provides the possibility to model systems with their influencing physical properties. A stepwise refinement of the system model has been introduced. One advantage is that the developed methodology is not orthogonal to the established standards MARTE or SysML and can therefore be easily adapted by these. The future work will cover a detailed discussion of the platform aspects and how domains are described at the component level. Another open issue is to go into deeper details of the physical and analog concepts of the methodology to better support the mixed-signal code generation and simulation.

## References

1. AUTOSAR. http://www.autosar.org/
2. Albers, K., Slomka, F.: Event Stream Calculus for Schedulability Analysis. In: Analysis, Architectures and Modelling of Embedded Systems, IFIP Advances in Information and Communication Technology, vol. 310, pp. 102–114. Springer Boston (2009)

3. aquintos: Preevision. http://www.aquintos.com/
4. Baumann, T., Salzwedel, H.: Mission Level Design using UML 2.0. In: Proceedings of the NODe '05, Object Oriented Software Design for Real Time and Embedded Computer Systems (2005)
5. Boulet, P., Marquet, P., Piel, É., Taillard, J.: Repetitive allocation modeling with marte. In: Forum on specification and design languages (FDL07) (2007)
6. Carloni, L., De Bernardinis, F., Pinello, C., Sangiovanni-Vincentelli, A., Sgroi, M.: Platform-Based Design for Embedded Systems. In: The Embedded Systems Handbook. R. Zurawski (Ed.) (2005)
7. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill Science/Engineering/Math (1994)
8. Gajski, D.: High-Level Synthesis. Kluwer (1992)
9. Gerstlauer, A., Shin, D., Peng, J., Domer, R., Gajski, D.: Automatic layer-based generation of system-on-chip bus communication models. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 26(9), 1676–1687 (2007)
10. Gladigau, J., Gerstlauer, A., Streubühr, M., Haubelt, C., Teich, J.: A System-Level Synthesis Approach from Formal Application Models to Generic Bus-Based MPSoCs. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) (2010)
11. International Society of Automation: ANSI/ISA-5.1-2009. http://www.isa.org/
12. Jacobson, I., Christerson, M., Jonsson, P.: Object- Oriented Software Engineering. Addison-Wesley Longman (1992)
13. Kollmann, S., Pollex, V., Kempf, K., Slomka, F., Traub, M., Bone, T., Becker, J.: Comparative Application of Real-Time Verification Methods to an Automotive Architecture. In: Proceedings of the 18th International Conference on Real-Time and Network Systems (2010)
14. Lee, E.: Cyber physical systems: Design challenges. In: 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC) (2008)
15. Liehr, A., Rolfs, H., Buchenrieder, K., Nageldinger, U.: Generating marte allocation models from activity threads. In: Forum on Specification, Verification and Design Languages (FDL08). pp. 215–220. IEEE (2008)
16. Matlab/Simulink. http://www.mathworks.de/
17. Münzenberger, R., Dörfel, M., Hofmann, R., Slomka, F.: A general time model for the specification and design of embedded real-time systems. Microelectronics Journal 34(11), 989–1000 (2003)
18. Object Management Group (OMG): Modeling and Analysis of Real Time and Embedded systems, version 1.1 (MARTE). http://www.omg.org/spec/MARTE/1.1/
19. Object Management Group (OMG): OMG Systems Modeling Language, version 1.2 (OMG SysML). http://www.sysml.org/specs/
20. Object Management Group (OMG): Unified Modeling Langauge (UML). http://www.uml.org/
21. Richter, K.: Compositional Scheduling Analysis Using Standard Event Models - The SymTA/S Approach. Ph.D. thesis, University of Braunschweig (2005)
22. Sangiovanni-Vincentelli, A., Martin, G.: Platform-Based Design and Software Design Methodology for Embedded Systems. In: IEEE Design and Test of Computers. vol. 18, pp. 23–33 (2001)
23. Schliecker, S., Hamann, A., Racu, R., Ernst, R.: Formal Methods for System Level Performance Analysis and Optimization. In: Proceedings of the Design Verification Conference (DVCon) (2008)
24. Sommerville, I.: Software Engineering. Pearson Studium (2001)
25. Teich, J., Haubelt, C.: Digitale Hardware/Software-Systeme. Springer (2010)
26. Traub, M.: Durchgängige Timing-Bewertung von Vernetzungsarchitekturen und Gateway-Systemen im Kraftfahrzeug. Ph.D. thesis, University of Karlsruhe (2010)
27. Wandeler, E.: Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems. Ph.D. thesis, ETH Zurich (September 2006)

# A Model-Driven Approach for Software Parallelization

Margarete Sackmann  Peter Ebraert and Dirk Janssens

Universiteit Antwerpen, Belgium
{margarete.sackmann, peter.ebraert, dirk.janssens}@ua.a.c.be

**Abstract.** In this paper, we describe a model-driven approach that aids the developer in parallelizing a software program. Information from a dynamic call tree, data dependencies and domain knowledge from the developer are combined in a model transformation process. This process leads to a model of the application where it becomes obvious which parts of the application can be executed in parallel.

## 1 Introduction

In the past, it was unnecessary to write programs that have the potential for parallel execution since most processors contained only one processing core and programs were therefore executed sequentially. Now, however, multiple processors are getting more and more popular, even for smaller devices such as routers or mobile phones, and it is crucial that the possibilities offered by the parallel processors are used well.

Despite this trend towards parallel processors, most programmers are still used to a sequential style of programming. It is difficult for them to judge where parallelization is both feasible and worthwhile. Even for programmers who are used to parallel programming, it is often less error-prone to start out from a sequential program rather than designing a parallel program from scratch, as errors that arise from the parallel execution can then be distinguished from other programming errors. In this paper, we propose a model-driven approach that aids the programmer in deciding where good opportunities for parallelization are in a program.

The starting point is the source code of a program from which a model of the behavior of the program is derived. This is done by profiling the execution of the program. In the resulting model, the developer may add information about the sequential order that may be required among program parts. Model transformations are then used to derive a parallel model of the application, taking into account both the information derived from profiling and the information added by the developer.

## 2 Related work

A popular way to parallelize programs is Thread Level Speculation (see for instance [1] or [2]). In this compiler-based approach, the program is parallelized during execution. The parallelization is speculative which means that in some cases a part of the execution has to be repeated since there are dependencies that were not taken into account. This approach seems unfit for systems where resources such as energy are scarce, since parts of the application have to be executed more than once to take effect.

Other approaches, for instance [3], require specific hardware. The speedups for floating point operations are close to optimal, however porting the application to different platforms is not possible.

When trying to parallelize applications for different hardware platforms, an approach that focuses on the software is a better solution. Examples are Embla 2 [4] which links back the profiling information of the program to the source code to help the user identify possibilities for parallelizing the code. iPat [5] is an Emacs assistance tool for parallelizing with OpenMP. However, it is entirely text-based and provides no visual aid to the developer.

In [6], a model transformation process is used to generate parallel code. The difference to our approach is that information about possible parallelism in the program has to be added to the code as compiler directives. Via various model transformations, code can then be generated for different hardware platforms.

An approach that helps to extract a parallel model of the application is the Compaan/Laura approach [7]. It gives a more complete view of the program than the approach presented here since the extracted model is representative for any input data. However, it requires a massive restructuring of the code and an initial idea about the program behavior to help with this restructuring.

## 3 Approach

We propose a toolchain that focuses on the software so as to ensure portability of the code to different hardware platforms. The approach is based on model transformations and provides a visual interface to the developer. Several models of the application are used throughout the toolchain to represent the application at different levels of abstraction and with different levels of detail. The advantage in this context is that the approach is not bound to the profiling tools we use and that details that are not important for the parallelization can be hidden. The resulting parallel model can be used to parallelize the source code or to schedule the application on a multiprocessor platform. Parts of the application are represented as nodes in a graph, and dependencies between them – implying precedence relations – as edges in that graph. Thus, the developer gets a visual aid that helps him to understand the data and control flow in the program. The toolchain will be illustrated on the Fast Fourier Transform (FFT) application of the MiBench benchmark [8].

In Figure 1, an overview of the transformation process from source code to parallel model is given. In the next section, the different models that are used throughout the transformation, i.e. call trees, call graphs and the parallel model, are introduced. In Section 5, the transformation from source code to a parallel model of the application is explained. Section 6 shows how to implement the model transformation in practice. The results of the transformation process for the FFT application are given in Section 7 and ideas for future work are explained in Section 8. We also explain some of the limitations of the approach, that can arise for instance from variable reuse or structural problems in the original program. Some conclusions are presented in Section 9.

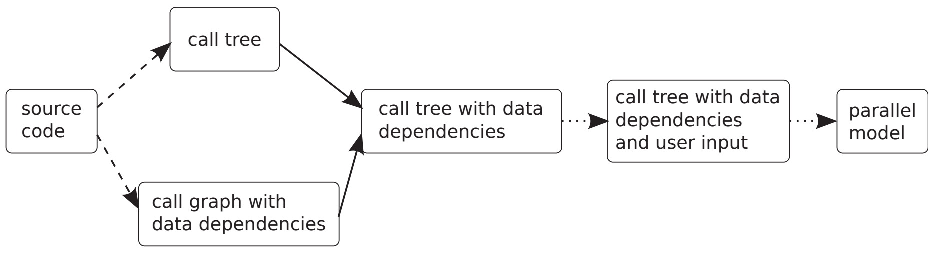

call tree
source code
call tree with data dependencies
call tree with data dependencies and user input
parallel model
call graph with data dependencies

**Fig. 1.** The transformation process. Dashed lines stand for the use of a profiler. Solid lines represent a parsing and transformation step. Dotted lines represent model transformations.

## 4 Models used during the Transformation

In this section we present the models that can be seen in Figure 1: the parallel model that is the goal of the model transformation, the call tree model and the enriched call tree model with data dependencies and dependencies added by the developer that is used during the model transformation.

To abstract from implementation details, we chose to represent the program at the level of functions. This means that a single function execution is the smallest part of the application that can be discerned in the different models. In case that is too fine-grained, several functions can be grouped into coarser-grained modules. If a finer grain is required, functions have to be broken up: If the user for instance suspects that a loop could be parallelized, then the calculations that are done within the loop body should be put into a separate function.

### 4.1 Parallel Model

The goal in the transformation process is to derive a model of the program that shows which parts of the application can be executed in parallel. The Parallel Model (PM) that is the outcome of the transformation process is a graph-based model. The nodes represent function instances that occur during an execution of a program. The edges in the parallel model represent a precedence relation between functions. An edge from function instance $f_1$ to $f_2$ means that $f_1$ has to start execution before $f_2$. The more important information in this model is however the lack of edges or a path between two functions. If there is no path between two function instances, they can be executed in parallel. This information can then be used for scheduling or to insert code to parallelize the program, for instance with OpenMP. A second type of edge, which we will call a subgraph edge, represents a stronger precedence relation. A subgraph edge from function $f_1$ to function $f_2$ means that $f_1$ and all its successors, i.e. its children, their children and so on, have to finish execution before $f_2$ can start execution.

The information provided by the subgraph edges can be used to get a greater flexibility of the model when the software has to be executed on different platforms with varying numbers of processors. In case the number of processors is small, a function $f$ that is the source or target of a subgraph edge can be combined with all its successors into a subgraph that is scheduled as a whole. For a larger number of processors, the subgraph edges can be replaced by regular edges and more nodes can be distributed over the parallel processors.

### 4.2 Call Trees

To derive the parallel model, we start by analyzing the program code. One way to analyze the relation between the different functions is to track the calls from one function to another. This can either be done statically on the whole source code, or dynamically at run time. We chose for dynamic analysis, since it represents the actual execution of the program. During program execution, a profiler tracks which functions are called in which order. The resulting model, the call tree, is specific to a single execution of the application. Therefore, this approach works best for applications that always have a similar behavior (i.e. the same function calls are executed even if the actual data on which the calculations are performed is different), for instance Digital Signal Processing (DSP) applications such as digital filters or matrix multiplications. An advantage of using dynamic call trees is that functions that are not used during execution (and thus do not have to be distributed on the parallel platform) are not considered. Recent research has shown that in many circumstances dynamic analysis is better suited for parallelization than static analysis (see for instance [9]). A disadvantage of using dynamic analysis is that programs may be executed differently based for instance on the input. In that case, a system scenario [10] approach, where different types of execution behaviors are identified can be used.

Call trees can be represented as graphs, where the nodes are function instances and the edges are calls between the function instances. An excerpt of the call tree that is derived from the FFT application can be seen in Figure 2.

As can be seen in Figure 1, besides function calls we also add data dependencies and dependencies added by the developer to the call tree in our toolchain. A data dependency edge from a function $f_1$ to a function $f_2$ means that $f_1$ reads or writes data that is also read by $f_2$. This implies that at least some part of $f_1$ has to occur before $f_2$. Otherwise, a correct outcome of the program cannot be guaranteed. $f_1$ and $f_2$ can consequently not be executed in parallel. At the moment, we use the Paralax [11] compiler to automatically detect the data dependencies. It logs for each data element that is used in the program, such as

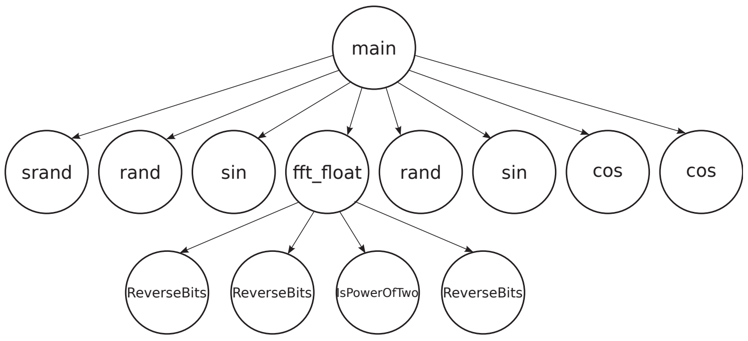


**Fig. 2.** Part of the call tree of the FFT application.

arrays or integer variables, which functions access this data in which order. This is done by tracking accesses to the address space of the data. Paralax also logs the access type, i.e. whether it is a read or write access. The read accesses are the critical ones and therefore the ones that are included into the model. The functions accessing a specific piece of data are identified by their call path, so that if a function is called from several other functions, a data dependency is only added if the whole call path matches. The data dependencies can be automatically included in the call tree model via a model transformation. To avoid loops, data dependencies are only added from a function $f_1$ to $f_2$ if $f_1$ occurred earlier in the call stack than $f_2$. The position in the call stack is recorded while obtaining the call tree and is a function attribute in the model transformation.

The third type of edge can be added by the user, in contrast to the function calls and data dependencies that can be detected and combined into a model automatically. The user can add edges between two function instances that are called by the same function instance. In Figure 2, the user can for instance add edges from the `srand` function instance to `rand`. An edge from $f_1$ to $f_2$ should be added if the execution of $f_1$ has to be finished before the execution of $f_2$ can start. Again, this implies that $f_1$ and $f_2$ cannot be executed in parallel. These extra edges can for instance be useful when the profiling tools could not detect all data dependencies due to the use of pointer arithmetics.

It could be argued that using call graphs rather than call trees would reduce the number of nodes. This approach is taken in [11], which we use to profile data dependencies. In a call graph, all instances of a function are combined into a single node. The problem with this approach is that the information contained in a call graph is not detailed enough. Consider for instance the call graph in Figure 3. The information that is included in this call graph is too limited to be able to parallelize the application properly. Many different executions can correspond to a call graph. In Figure 4 two call trees are shown that would both result in the call graph of Figure 3. For the parallelization step that is described in the following chapter it is important however, that the exact instance of a function that calls another function is known.

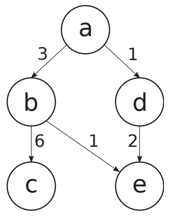
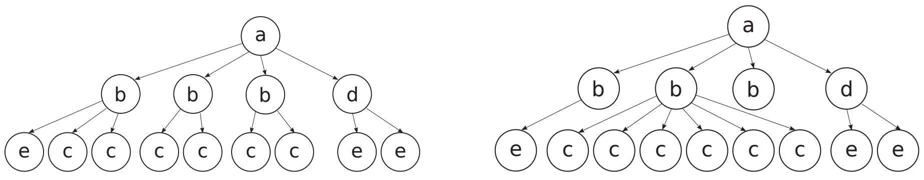


**Fig. 3.** A call graph



(a) Evenly distributed call tree

(b) One instance of $b$ calls all instances of $c$

**Fig. 4.** Two possible call trees for the call graph in Figure 3

## 5 Transforming Source Code into a Parallel Model

In this section, the transformation process from the source code of an application to a parallel model of this application is described. The starting point is a dynamic call tree of the application (see 5.1) that is then enhanced with additional edges that represent data dependencies (5.2) and domain-specific knowledge of the user (5.3). The automatic transformation into the parallel model is described in Subsection 5.4. In Subsection 5.5, the usage of special edges that represent a dependency between subgraphs in the parallel model is explained.

We assume that, if a function $f$ calls a function $g$, then there are no further computations (only possibly calls to other functions) in $f$ after $g$ returns. This form can be easily established in the source code: in case there are computations in $f$ after the call to $g$, then these can be put into a new function $h$, and a call to $h$ is added in $f$ after the call to $g$.

### 5.1 Pruning the Call Tree

As explained above, the size of a call tree can be quite large. Even for simple programs, thousands of functions are called. However, many of these functions are library calls or other functions that the user does not have control over. Therefore, we concentrate on functions that either are part of the source code that the developer programmed, or that are called from within that source code.

*Example 1.* In Figure 5 a loop from the `main` routine of the FFT application is shown. The functions `sin` and `cos` are called inside a loop. Even though these functions were not implemented by the user, they can be responsible for a large amount of execution time. Additionally, the user can add code to parallelize the different calls to these functions. Therefore, these functions are included in the call tree, but the functions that are not visible in the source code are not included (for instance functions that are called by `sin`).

```
for(i=0;i<MAXSIZE;i++) {
    RealIn[i]=0;
    for(j=0;j<MAXWAVES;j++) {
        if (rand()%2) {
            RealIn[i]+=coeff[j]*cos(amp[j]*i);
        }
        else {
            RealIn[i]+=coeff[j]*sin(amp[j]*i);
        }
    }
}
```

**Fig. 5.** Standard library functions called within a program

Even when leaving out calls to library functions, the number of function instances can be overwhelming. Therefore, only the functions with the largest instruction count are picked. Although this instruction count can vary on some processors with specialized computation units and may also be different for different compilers, this number is in general a good indicator on the expected execution time. This way, the user can later concentrate on parallelizing those parts of the software that make up the largest part of the execution time.

### 5.2 Data Dependencies

Besides the calling relation between functions, there exist other relationships between functions that can require them to be executed sequentially. If two functions for instance change the value of the same variable, then they have to be executed in a given order. However, these data dependencies are not visible in the call tree. Adding data dependencies that are later taken into account when producing the parallel model prevents race conditions that might occur when the program is parallelized.

As we work with models of the application, adding precedence relations between functions simply means adding directed edges in the graph representation of the model. Therefore, different profilers can be used to derive the data dependencies. Even profilers that can extract different dependencies between function instances could be used to include more edges in the call tree.

## 5.3 User Input

Since it is possible that some dependencies that will prohibit a parallelization are not included in the dependencies from the call tree and the data dependencies, the developer gets to add precedence relations that were not discovered by the profilers. Sometimes, as in the case of the FFT application, not all dependencies can be recognized by the profiler for instance due to the use of pointer arithmetics. In that case, the user has to add these dependencies by hand. Simply allowing the user to add edges between any two nodes of the graph would, however, not result in a very good parallel model. A call tree can be rather large and the user is bound to make mistakes if he is allowed to add edges unrestrictedly. In addition, parallelizing function instances that are called from the same function can be realized rather easily and without having to rewrite parts of the code. Therefore we chose a step-by-step approach, where only a part of the call tree is presented to the user in each step.

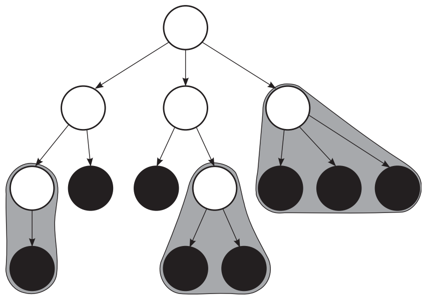


**Fig. 6.** A call tree where all leaves are shown in black. The subtrees marked in gray consist of one parent node and its children nodes that all have to be leaves.

A subgraph that consists of a parent node and its children nodes is presented to the user in each step. The children nodes all have to be leaves of the call tree, as illustrated in Figure 6. If there is a dependency between two children nodes of the subgraph, the user adds an edge between these nodes. In case that the user does not add any precedence relations for some children, it is assumed that these children nodes can be executed in parallel.

### 5.4 Transformation to a Parallel Model

In the automatic step from the enhanced call tree to the parallel model of the application, all the information from the profilers and the developer is combined. A model transformation is used to build up the parallel model of the application, while at the same time deleting nodes from the call tree in a bottom-up way. Before presenting the first subgraph to the user, all nodes of the call tree are copied to a new graph, the parallel model $PM$, without however copying the edges. The edges are inserted step-by-step, as the subgraphs are modified by the user. Redundant edges are not copied to the parallel model. An edge between nodes $f_0$ and $f_1$ is redundant if there is a directed path from $f_0$ to $f_1$ that consists of edges of the same or a higher importance. The call tree edges are the least important edges and the data dependencies are the most important edges.

We chose to keep all data dependency edges in the transformation from the call tree to the parallel model. This is not strictly necessary to get a parallel model of the application. However, it is better to keep the data dependencies intact if this parallel model is used for scheduling where the data communication is important, such as embedded systems. If the data on a data dependency edge is for instance very large, it is better to schedule the functions on both ends of this data dependency on the same processor to avoid a large overhead for communication.

After copying the edges of the call tree subgraph in the manner just described to the parallel model, all children nodes of the subgraph are now deleted in the call tree. That way, it is ensured that for each function $f$, the subgraph with $f$ as the parent node is presented to the user at some point.

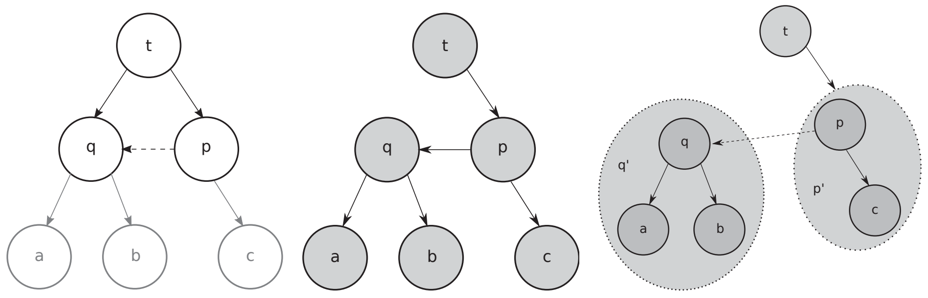
### 5.5 Subgraph Edges

In the parallel model $PM$ one needs to be able to see which subgraphs were already considered, i.e. for each node it needs to be obvious which function instance it was called from. This is best explained by way of an example.

*Example 2.* Imagine a subgraph with parent node $p$ that has a child $c$. According to the transformation algorithm, the edge from $p$ to $c$ is copied to $PM$. In a later step, $p$ and another node $q$ are the children in a subgraph with parent $t$ as shown in Figure 7(a). The user now indicates that $q$ has to be executed after $p$ as indicated by the dotted line in the figure. If no information is kept that $p$ and $c$ formed a subgraph earlier, then edges will now go from $p$ to both $c$ and $q$ in the parallel model, indicating that $c$ and $q$ can be executed in parallel. This is shown in Figure 7(b). However, this is impossible to implement in practice without rewriting a part of the code.

This is why subgraph edges are allowed in the parallel model. These edges can be removed in a later step and replaced with the regular edges in the parallel model, but are useful during the transformation phase. For the example that was just discussed, the parallel model in Figure 7(c) is the result of using a subgraph edge. All nodes inside the dotted circle on the right $p'$ have to be executed before all nodes inside the dotted circle $q'$ on the left. The subgraph edge is indicated by a dashed arrow.

We could of course also allow users to specify that $c$ and $q$ in the example in Figure 7 can indeed be executed in parallel. However, in practice this will most likely be difficult to realize. It would force the programmer to ensure that $q$ can be executed in parallel with $c$, but only after some of the computations of $p$ are already done. While our approach is more conservative, it allows for an easier adaptation of the source code and will lead to less errors.



(a) A call tree with a precedence relation from $p$ to $q$. The parts that are in gray are already deleted from the call tree, but are shown here to indicate the precedence relations in the subtrees with parents $p$ and $q$.

(b) Transforming the call tree into a parallel model without subgraph edges. Here, it seems as if $q$ and $c$ could be executed in parallel.

(c) Transforming the call tree into a parallel model using a subgraph edge. In this case, all parts of $p'$ are executed before any nodes in $q'$ can start execution.

**Fig. 7.** Using subgraph edges to prevent undesired parallelism in the parallel model after the transformation from a call tree

Subgraph edges can e.g. be useful if the parallel model is used for scheduling, but the number of nodes in the call tree is much too large compared to the number of processors. In that case, subgraphs can be seen as a single node.

An optional model transformation step allows to replace subgraphs and subgraph edges with nodes and regular edges to adapt the granularity of the parallel model.

*Example 3.* An example is given in Figure 8, where $h$ represents the subgraph that consists of $h_0$ and its successors. The edges that lead away from $h_0$ have to be redirected. For that, all leaves within $h$ have to be found. In Figure 8 this would be nodes $h_1$, $h_3$ and $h_4$ since they have no outgoing edges. The subgraph edges going out from $h_0$ will each lead to three outgoing edges from $h_1$, $h_3$ and $h_4$ respectively. Finally, the subgraph edges can be deleted. The result of replacing the subgraph edges of $h_0$ in Figure 8 is shown in Figure 9.

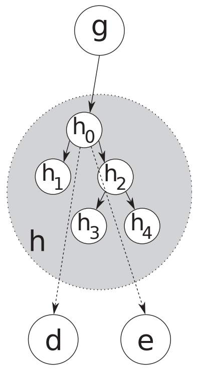


**Fig. 8.** Subgraph edges in the parallel model

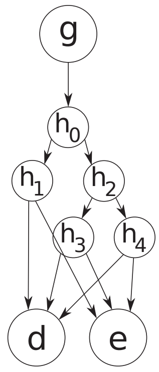


**Fig. 9.** Replacing the subgraph edges in Figure 8

## 6 Implementation

We implemented the above approach in a toolchain that goes from the software source code via call trees to a parallel model of a software application.

The call tree is represented graphically within the AToM$^3$ [12] modeling tool. Each edge type – function call, data dependency and user-defined dependency – has a different color, so that the user can keep track of what kind of dependency an edge represents.

The first type of edge is a function call. To derive these edges, the toolchain uses a modified version of the Callgrind profiler [13] that is part of the Valgrind tool suite [14]. The adaptations were necessary to filter out the functions that do not appear in the source code of the application. In addition, Callgrind had to be modified to produce a call tree as an output rather than a call graph. The call tree that is produced as the output of the modified Callgrind tool is then parsed into a Python file that can be opened in AToM$^3$. In the parsing step, we allow to adapt the granularity of the call tree by transferring only the function instances with the largest instruction count to the Python file.

The second type of edges are data dependencies. They are derived from the output of the Paralax [11] compiler. Data dependencies in Paralax are dependencies between different call paths. For a data dependency between call path $f_1, \ldots, f_n$ and call path $g_1, \ldots, g_m$ that is found by the Paralax compiler, an edge has to be inserted in the call tree between each instance of $f_n$ and each instance of $g_m$ in the call tree, provided that the predecessors of $f_n$ resp. $g_m$ are instances of $f_1, \ldots, f_{n-1}$ resp. $g_1, \ldots, g_{m-1}$.

*Example 4.* Consider the call tree in Figure 10. Assume that there is a data dependency from call path $a \to b$ to $a \to c \to d$. Then, two data dependencies would be added, from $b_1$ to $d_1$ and $d_2$. There will be no data dependency from $b_1$ to $d_3$, since the call path of $d_3$ does not match with the one described by the call path $a \to c \to d$. The resulting call tree with the added data dependencies is shown in Figure 11.

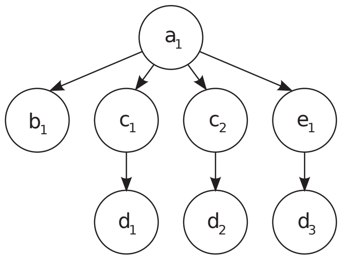


**Fig. 10.** A call tree where function $d$ can be called from functions $c$ and $e$.

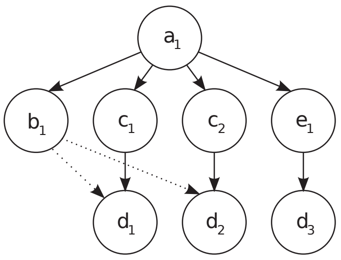


**Fig. 11.** Adding the data dependency from call path $a \to b$ to $a \to c \to d$ in Figure 10.

The last type of edge is a user-defined edge. As explained above, these user-added edges can be added one subgraph at a time. In each step, a subgraph of the call tree is highlighted. This subtree consists of a function instance $p$ and all its children that all have to be leaves in the call tree. The user can add missing dependencies by drawing edges in the model. The position of each function instance in the call stack is given, so that the developer can see in which order the different function instances were actually executed during the profiling.

Once the user inserted all edges in a subgraph, he can indicate this with the click of a button and wait until the next subgraph is highlighted. The edges of the current subgraph are then copied to the parallel model in the manner described above. All children of the subgraph are then deleted along with the edges leading to and from them. A rule that determines for each node if it is a leaf of the call tree is invoked, so that the parent of the call tree will now be marked as a leaf. Then, the next subgraph is highlighted and so on.

After all nodes of the call tree have been deleted there is an optional step for replacing the subgraph edges. This is done in a top-down way, meaning that the subgraph edges closest to the root node (usually the `main` function) are replaced first so that the granularity of the model can be adapted.

## 7 Results

We tested our approach on the FFT implementation of the MiBench benchmark. In order to get usable results, we had to divide the loop that computes the FFT in the program into function calls. Although this was done within a few minutes, it shows a limitation of our approach that is due to it using functions as the smallest entities.

During the transformation phase, it was shown that it is a good idea to present the model to the user instead of just relying on the information given by the profiling tools. When we profiled the FFT application, the dependency that arises from filling the input signals with random numbers as seen in Figure 5 to using these signals in the actual Fast Fourier Transform was not detected by the Paralax profiler.

From the resulting parallel model, it became apparent that the best opportunities for parallelization are in two loops. The first loop is the inner loop of the actual Fast Fourier transformation (the outer loop has data dependencies) and the second loop is the one shown in Figure 5. These two loops make up more than 50 percent of the execution time. This shows that our approach can indeed help the developer in identifying the program parts where parallelization will bring the largest benefit.

The code was parallelized by inserting OpenMP [15] directives according to the parallel model. To parallelize the loop that computes the FFT, it was enough to add a simple line of code that allows the concurrent execution of the loop. Parallelizing the loop that initiates the input signals for the FFT is more difficult. As seen in Figure 5, there is a conditional execution within the loop. Therefore, the code had to be changed into the form seen in Figure 12. This already gives a slight improvement in the sequential execution of the code. This new for-loop can then be parallelized. Note that in order to get the same results as with the sequential version of the code, it has to be ensured that the random numbers are used in the same order. Therefore, they are stored beforehand in a separate array. Creating this array takes about one quarter of the total execution time of the parallelized program. When executing the program on an Intel Core i3 processor (dual core model with hyper-threading technology), the overall speedup when parallelizing both loops in the manner just described was 2.5.

```
for(i=0;i<MAXSIZE;i++) {
  RealIn[i]=0;
  for(j=0;j<MAXWAVES;j++) {
    x = rand()%2;
    RealIn[i] += x*coeff[j]*cos(amp[j]*i)+
                 (1-x)*coeff[j]*sin(amp[j]*i);
  }
}
```

**Fig. 12.** Modifying the code in Figure 5 so that there is no conditional statement in the for-loop.

## 8 Future Work

A line of future work would be to automatically trace back the parallelism that becomes apparent in the parallel model to the source code (as for instance seen in [16]). This could then be used to include the necessary code that parallelizes the application automatically. However, since the correctness of the parallel model depends on the user input, the model is not guaranteed to be correct. It might be more difficult to remove automatically added code that is incorrect than to add the code. Additionally, as can be seen from the FFT example, parallelizing the code can sometimes mean that some part of the code has to be rewritten before parallelizing into different threads can have a benefit. Another problem with automatically generating code is variable reuse. In case that during sequential execution a variable is used in a part of the code and then again in a different part of the code in a different context, these two parts could be executed in parallel (there will only be a write-after-read dependency, which will not be included in the model transformation). However, to actually implement this parallelism two distinct variables are needed. In order to be able to do that automatically, the write-after-read dependency would have to be tracked.

When testing our approach on a medical imaging software, it became apparent that pipeline parallelism is difficult to detect with our approach. In the case of the medical imaging software, a 4-stage pipeline is used. Each pipeline stage writes on an array that is then read by the next stage. Since the data dependency profiler does not distinguish between function instances, dependencies are added from each stage 1 function instance to each stage 2 function instance that occurred after it in the call stack. This makes the model very cluttered and the developer might not be able to see the possible parallelism.

The approach can be easily extended to include other profiling information about dependencies between different functions. Our approach already considers dependencies that can be left out in the parallel model if alternative paths exist in the model (the function calls), and dependencies that are always copied from the original call tree to the parallel model. Therefore, it would be straightforward to include other dependencies without having to adapt the graph transformation algorithm.

## 9 Conclusion

In this paper, we proposed a model-driven approach to help developers parallelize sequential software programs. Profiling results are combined with domain-knowledge of the user to transform source code into a parallel model of the application. We believe that by abstracting away the implementation details we can assist the developer significantly in the parallelization process. By providing the developer with an abstract model of the software that already includes function calls and data dependencies, the developer gets a graphical representation of his program that hides irrelevant details. As the developer can add precedence relations, a parallel model can be derived. It can for instance be used for scheduling the application on a multi-processor platform or adding compiler directives for parallel execution into the source code.

## References

1. Quiñones, C.G., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.M.: Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. (In: 2005 Conference on Programming Language Design and Implementation (PLDI))
2. Steffan, J.G., Colohan, C., Zhai, A., Mowry, T.C.: The STAMPede Approach to Thread-Level Speculation. Transactions on Computer Science (2005)
3. Chen, M.K., Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs. (In: 30th Annual International Symposium on Computer Architecture (ISCA '03))
4. Mak, J., Faxèn, K.F., Janson, S., Mycroft, A.: Estimating and exploiting potential parallelism by source-level dependence profiling. (In: Euro-Par 2010)
5. Ishihara, M., Honda, K., Sato, M.: Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP:iPat/OMP. IEICE Transactions on Information and Systems (2006)
6. Ismail Assayad, Valérie Bertin, F.X.D.P.G.O.Q.S.Y.: Jahuel: A formal framework for software synthesis. In: Proceedings of the Seventh International Conference on Formal Engineering Methods ICFEM. (2005)
7. Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.: System design using kahn process networks: The compaan/laura approach. In: Proceedings of the conference on Design, Automation and Test in Europe (DATE). (204)
8. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. (In: Proceedings of the Workload Characterization 2001 Workshop)
9. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization. (In: 2009 Conference on Programming Language Design and Implementation (PLDI))
10. Miniskar, N.R., Hammari, E., Munaga, S., Mamagkakis, S., Kjeldsberg, P.G., Catthoor, F.: Scenario based mapping of dynamic applications on mpsoc: A 3d graphics case study. In: SAMOS Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling and Simulation. (2009)
11. Vandierendonck, H., Rul, S., de Bosschere, K.: The paralax infrastructure: Automatic parallelization with a helping hand. In: Parallel Architectures and Compilation Techniques (PACT). (2010)
12. de Lara, J., Vangheluwe, H.: Using AToM3 as a Meta-CASE Tool. (In: 4th International Conference on Enterprise Information Systems (ICEIS 2002))
13. Weisendorfer, J., Kowarschik, M., Trinitis, C.: A tool suite for simulation based analysis of memory access behavior. (In: Proc. of the 4th Int. Conference on Computational Science (ICCS 2004))
14. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instructions. (In: Proc. of ACM SIGPLAN 2007 Conference on Programming Language and Design (PLDI 2007))
15. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP. The MIT Press (2007)
16. Johnson, S., Evans, E., Ierotheou, C.: The parawise expert assistant – widening accessibility to efficient and scalable tool generated openmp code. (In: Workshop on OpenMP Applications and Tools (WOMPAT 2004))

# A Refinement Checking Technique for Contract-Based Architecture Designs *

Raphael Weber[1], Tayfun Gezgin[1], and Maurice Girod[2]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany,
{raphael.weber,tayfun.gezgin}@offis.de
[2] Airbus Operations GmbH, Kreetslag 10, 21111 Hamburg, Germany,
maurice.girod@airbus.com

**Abstract.** During the development of software intensive systems, typically several models of this system are designed. These various models represent the system structured by different concerns, e. g. abstraction. While these approaches help to cope with complexity, the need of relating the models to one another arises. A major task is to keep model specifications consistent and traceable through special relations. The relation of interest for this work is the refinement relation between abstraction levels. In this work we describe a technique to check the validity of these refinement relations with respect to formal behavior/interface specifications of design items. For evaluation, we apply our refinement technique to an industrial example modeled with the contract-based methodology from our previous work.

## 1 Introduction

In recent years, the design process of systems in domains like automotive, automation technology, avionics, or consumer electronics has become a more and more complex task: The increasing number of functions which are realized by software, inter-dependencies of software tasks, and the integration of existing sub-systems lead to highly complex software intensive systems. This complexity in conjunction with the increasing demand on a short time-to-market and on quality aspects make it difficult for engineers to develop such systems. In order to cope with this difficulty we proposed a new meta-model along with a methodology in [2] to support the system architect.

The proposed common systems meta-model (CSM) is based on the meta-model of *Heterogeneous Rich Components* (HRC) [6], which provides the concept of contract-based specifications. The term “rich” alludes to the key ingredient of HRCs to provide rigorous interface specifications for multiple *aspects*, encompassing both functional and extra-functional (e. g. safety and real-time) characteristics of components. To structure the design space and enable different development approaches (e. g. top-down, bottom-up, . . . ), we described the

* This paper was partly funded by the German Federal Ministry of Education and Research (BMBF) through the project “Software Platform Embedded Systems (SPES 2020)”, grant no. 01IS08045.

concept of *Abstraction Levels* and *Perspectives* along with their CSM representation. For the designer to navigate and model relations between abstraction levels and perspectives, we briefly introduced the concept of a *Mapping* relation (called *Realization* between abstraction levels and *Allocation* between perspectives). Analyzing these mappings was not discussed in detail in our previous work and is subject of this work.

For evaluation, an industrial example was proposed by Airbus: the Air-Conditioning-System. We evaluate a certain portion of that example through refinement check. This check finds out whether a component of a specific design can be replaced by a component of a different design, i. e. it finds out whether the different design component can be virtually integrated into the environment of the other component. The refinement check is done via model checking an Uppaal [9] timed automata representation of the formal requirements of the model.

By formal requirements we mean the specification of requirements via the pattern-based requirements specification language (RSL). The RSL is developed and evaluated in the European CESAR project. While this requirement specification technique allows for easy transformation from predefined patterns to automata it may not be the most intuitivly usable one. Yet, there are a variety of methods to help to derive natural language requirements down to pattern based requirement specifications (contracts). However, that subject is not part of this paper, for more information about the RSL and requirement specification methods in general refer to [4].

**Related Work** In current literature one can find many works that deal with refinement checks via timed automata. One similar work is [15] in which the authors use Uppaal timed automata (among others) to verify the correctnes of their requirements. However, timed automata are not used to formally check the refinement of these requirements. The theoretical foundation and practical application of the contract-based specification method were elaborated in [5, 3]. One could consider our work as a follow-up work of [5]. In [13] the authors propose a basic calculus for adding and removing channels and components in a dataflow architecture. The calculus formally allows for refinement checking between two system architectures. While this approach covers adding and removing entities it does not include the modification of artifacts. Furthermore, there is no relation to meta-modelling concepts or to the (then emerging) UML standard. There is also no practical example on which the calculus was applied and evaluated. There is, however, an introducing example to motivate the approach.

The remaining parts of this paper are organized as follows. In the next section we give a short introduction on the modeling concepts of the CSM. In Section 3 we describe the refinement check for the case in which one contract is refined by exactly another contract and give the construction principle of the automaton networks. Section 4 illustrates the air-conditioning-system example by Airbus and explains how we applied the refinement check to the example. In Section 5 we discuss the extension of the refinement check where the specification of a component may consist of a set of contracts. In the last section we will sum up our findings and draw a conclusion.

## 2 Modeling Concepts

In our previous work [2] on a new common systems meta-model (CSM), Heterogeneous Rich Components (HRCs), which originate from the European SPEEDS project [17], represent the major modelling artifact. In addition to HRCs a new methodology to traverse along the design space is also proposed in [2]. This section gives a short introduction to the concepts of HRC and the design methodology of the CSM.

### 2.1 Heterogeneous Rich Components

The CSM provides basic constructs needed to model systems like components with ports and connections (bindings) between them. We refer to these components as Heterogeneous Rich Components (HRCs). HRCs rely on the basic concepts of SysML blocks [12]. The dynamics of an HRC can be specified by behavior, e. g. an external behavior model, or even source code. Furthermore, requirements (or contracts) refer to a *required* behavior whereas the *actual* behavior is specified as stated above. The idea of contracts is insprired by Bertrand Meyer's programming language Eiffel and its *design by contract* paradigm [10].

In HRC, contracts are a pair consisting of an assumption and a guarantee, both of which are specified by some text. Here we assume that assumptions and guarantees are specified by a pattern based formalism called requirement specification language. An assumption specifies how the context of the component, i. e. the environment from the point of view of the component, should behave. Only if the assumption is fulfilled, the component will behave as guaranteed. This enables the verification of virtual system-integration (integrate a more detailed component or a subcomponent in a more abstract environment) at an early stage in a design flow, even when there is no implementation yet. Thus, the system decomposition can be verified with respect to contracts. Details about the semantics of HRC are given in Section 3.1. Note that in this work we consider only the HRC semantics where a connection between two ports describes their equality. A deeper insight into HRCs can also be found in [8, 16].

### 2.2 Structuring the development process

When developing an embedded system, an architecture is regarded in different *Perspectives* at several *Abstraction Levels* during the design process as mentioned in Section 1. On each abstraction level the product architecture is regarded in different perspectives. As an additional concept for separation of concerns, models in each perspective reflect different aspects. For example, an aspect "Safety" might be regarded in every perspective but an aspect "Realtime" is not regarded in a geometric perspective and aspect "Cost" is not regarded when considering operational use cases.

### 2.3 Realization and Allocation

In order to keep the models in different perspectives and abstraction levels consistent (keep traceability between development steps) we defined a so called *Realization-* and *Allocation-Link*. The basic idea behind both concepts is to relate the observable behavior of components exposed at its ports.

Realizations are relationships between ports of components on different abstraction levels. Intuitively a realization-link states, that a component (e. g. *f1*) has somehow been refined and is now more concrete in terms of its interface and/or behavior (e. g. *f1'*). The refinement cannot always be captured by a pure decomposition approach. Thus, we define the realization of a component by introducing a state-machine that translates the behavior of the refined component *f1'* into according events observable at a port of component *f1*.

Allocations are relationships between ports of components in different perspectives. Intuitively an allocation-link states that the logical behavior of a component (e. g. *f4*) is part of the behavior of a resource (e. g. *r2*), to which it has been allocated. Here, we consider the same link-semantics as for the realization. For more details, refer to [2]. In the next section we will deal with the automatic verification of allocation and realize links in more detail.

## 3 Refinement Check

In the previous section we introduced our methodological concepts of the CSM and the HRCs. This section describes how the concepts of realize and allocate links can be automatically validated. For this, we first give an in-depth description of the semantics of HRCs.

The concepts of realize and allocate links between perspectives and abstraction levels are very similar. Both links define a refinement relation between components with respect to their specification: The refined components have to respect the requirements specified for their abstract counterparts. We say, a specification $C'$ refines another specification $C$ if and only if the behavior specified by $C'$ is a subset of the behavior specified by $C$. In the following we will formalize the refinement relation with respect to contracts and give a technique in order to automatically check such a relation. In this work we will only consider 1-to-1 mappings, i. e. where a component is refined by exactly another component. Mappings, where a set of components is related to another set of refined components is subject of future work. Furthermore, we will only consider specifications which consist of exactly one contract. The generalization to a set of contracts is subject of Section 5.

### 3.1 Semantics of HRC

The specification of HRCs is given in terms of contracts over their interaction points capturing the required dynamics of a component. This means that for specifically assumed environment conditions the component shall guarantee a specific behavior exposed at its ports. In the following we assume without loss of generality that each port contains exactly one interaction point, so we can refer to that interaction point when talking about a port.

A contract is a tuple $(A, G)$, where $A$ defines the assumption and $G$ the guarantee as introduced in Section 2. The semantics of a contract is defined as

$$[\![C]\!] := [\![A]\!]^{Cmpl} \cup [\![G]\!], \tag{1}$$

where $(X)^{Cmpl}$ defines the complement of a set $X$ in some universe $\mathcal{U}$ and $[\![X]\!]$ is defined as the semantic interpretation of $X$.

The semantics of $A$ and $G$ is given in terms of sets of timed traces. A trace over a port set $P$ is a sequence of port assignments. A port assignment $\mathcal{V}$ is a function $\mathcal{V}: P \to D$ assigning each port $p_i \in P$ to a value in its domain $D_i$. Further, a time sequence $\tau$ is a monotonically increasing sequence of real values, such that for each $t \in \mathbb{R}$ there exist some $i \geq 1$ such that $\tau_i > t$. A timed trace is a sequence $(\rho, \tau)$ where $\rho$ is a sequence of port assignements and $\tau$ a time sequence. The set of all timed traces over $P$ is denoted by $Tr(P)$.

The specification $S$ of a component is given in terms of a set of contracts, i. e. $[\![S]\!] := \bigcap_{i=1}^{n} [\![C_i]\!]$. An implementation $I$ of a component satisfies its specification $S$, if $[\![I]\!] \subseteq [\![S]\!]$ holds. The refinement relation between two contracts $C$ and $C'$ is defined in a similar way. Note that the definition for $n$ contracts is subject of Section 5.

$$C' \text{ refines } C, \text{ if } [\![A]\!] \subseteq \alpha([\![A']\!]) \text{ and } \alpha([\![C']\!]) \subseteq [\![C]\!], \tag{2}$$

where $\alpha : Tr(P') \to Tr(P)$ is called the mapping function (represented through a state machine as mentioned in Subsection 2.3) relating concrete traces with abstract ones. This function is necessary as both contracts $C$ and $C'$ may talk about different ports. Here we only assume mapping functions which can be transformed to timed automata. Note that mapping functions are not generated automatically but rahter are given by the systems architect.

In the following subsection we will introduce a technique to check such a refinement relation.

## 3.2 Checking the Refinement Relation

We introduce our concept of verifying a refinement relation by specifications consisting of only one contract. In order to check the refinement relation between the contract of the abstract and the concrete component, we derive Uppaal timed automata[9] out of both contracts and do a reachability check[1]. As defined in Equation 2 the check consists of two parts, i. e. first checking the set inclusion of the assumptions and second checking the set inclusion of both contracts.

**Checking Set Inclusion of Assumptions** In order to check $[\![A]\!] \subseteq \alpha([\![A']\!])$ we derive a timed automaton out of $A'$ serving as passive observer $O$. The transitions of $O$ are annotated with receiving events (derived out of port names) and clock constraints in such a way, that the observer accepts the set of timed traces which are element of $[\![A']\!]$. For all traces which are not element of $[\![A']\!]$ the observer enters a bad state. Then we derive an automaton $T_A$ out of $A$ which serves as a trigger for $O$. The transitions of $T_A$ are annotated with sending events and timing constraints, such that $T_A$ produces all traces that are element of $A$.

At least we need the automaton $Gl$ realizing the mapping function $\alpha$ which receives events from $T_A$ and translates corresponding events to the observer. This automaton is assumed to be given by the system architect.

From all automata we build the automaton network $T_A \parallel Gl \parallel O$. If the trigger automaton now produces a sequence which is not element of $A'$ — and therefore the subset inclusion property will be violated — the corresponding observer will enter a bad state. So we need to check $T_A \parallel Gl \parallel O$ against the following Uppaal query: $A\Box$ *not* $O.bad$.

**Checking Set Inclusion of Contracts** The second part consists of checking $\alpha([\![C']\!]) \subseteq [\![C]\!]$. This is done by deriving an automaton network $T_{C'}$ out of $C'$ or more precisely one automaton for the assumption part and one for the guarantee part. Both automata serve as trigger for the observer network derived out of $C$. Analogously to the first part it holds that whenever one of the trigger automata does a step which is not defined in the contract of the abstract component, the corresponding observer will enter a bad state.

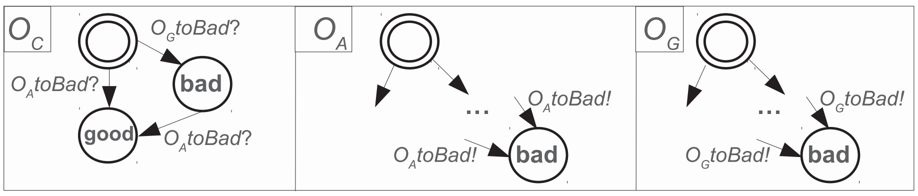


**Fig. 1.** Automaton network for Observer: $O_G$ is the automaton for the guarantee, $O_A$ for the assumption and $O_C$ for the overall state of the contract.

The observer consists of three automata: one automaton for each assumption and guarantee and a third automaton tracing the state of the overall contract. This is illustrated in Figure 1: The observer obtained from the guarantee ($O_G$) and the assumption ($O_A$) send an event to the automaton $O_C$ when entering their bad state. The automaton $O_C$ states whether the contract is fulfilled. According to Equation 1 this is the case when either both the assumption and guarantee are fulfilled or the assumption does not hold. If the assumption does not hold, $O_A$ sends an event $O^A_{toBad}$ such that $O_C$ directly switches to state `good`. If the guarantee was not fulfilled previously, then $O_C$ switches to its good state in the case that the assumption is finally violated.

The automaton network $T_{C'} \parallel O_G \parallel O_A \parallel O_C$ is again checked against the Uppaal query: $A\Box$ *not* $O_C.bad$. If $O_C$ enters its bad state as the corresponding guarantee is not fulfilled, we need to check whether finally its assumption automaton also enters its bad state, such that $O_C$ switches to its state *good*. This is realized by checking the query $O_C.bad \dashrightarrow O_C.good$. The arrow is the *leads-to* operator of Uppaal and states for this case that whenever $O_C$ enters its bad state it will finally enter its good state.

## 4 Case Study: Air-Conditioning-System

In this section we will describe in text and figures how the air-conditioning-system was modeled. Note that this is only a cut-out on two abstraction levels described in detail in Subsections 4.2 and 4.3.

### 4.1 Preliminary Notes on the Example

The following two subsections contain details about a cut-out of an example model of the air-condintioning-system (ACS) we did with Airbus in the scope of the SPES2020 project. This cut-out deals only with the technical perspective on two different abstraction levels, called the "upper" and the "lower". In each technical perspective we first define our system context (the environment outside the system under development). The boundaries of this environment description may change between abstraction levels, as is the case in our example. Note that we renamed some entities of the model.

The *air-generation demand calculation system* which is part of the ACS was previously modeled in the logical perspective (not described here). It is subject of the upper level whereas the processor of the upper level is refined in the lower level. The contracts of this particular processor and its subcomponents are described and examined for a refinement check which is described in Subsection 4.4.

### 4.2 The Upper Abstraction Level

The air-generation demand calculation system is designed in a rather fine level of granularity. At this point in the design process, it is time to initially design the technical architecture. Our system context contains the air-generation demand calculation system with four inputs (selected air demand from two crew selections and two sensor values of the actual climate to be conditioned) and one output (the calculated air demand value).

The system under design (the air-generation demand calculation system) is decomposed into certain resource artifacts. The *Input_ Terminal* (a device resource, fetching data from certain memory addresses), the *Input_ Com* (a communication resource, among others, sending requested data over an Input bus to a computation resource), the *CPU1* (a computing resource on which the actual air-generation demand calculation task is executed), the *Out_ Com* (a communication resource which, among others, sends the result of the calculation to the Out_Terminal, to be distributed), and the *Out_ Terminal* (a device resource, writing data to certain memory addresses). The schedule, according to which the task is scheduled, is a static time table, just like an ARINC653-Module specification (see [7]). The system under design with its artifacts is displayed in Figure 2.

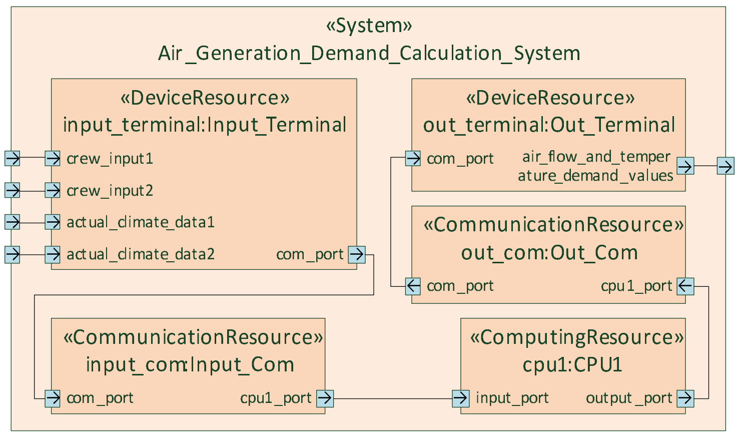


**Fig. 2.** The air-generation demand calculation system with resources.

The terminals and busses details are not described here. Instead we focus on the model of the *CPU1*. It contains one task (*demand_ calculation*), which performs the actual air flow and temperature calculation based on its inputs and writes it to its output port. Furthermore, the *CPU1* component contains a scheduler slot (which contains information, relevant for the scheduler) representing a certain amount of time within the schedule of *CPU1*. *CPU1* also contains the scheduler that has a certain scheduling policy. The ports between the slot and the task are SchedulerPorts, over which the slot "tells" the task when it is executing. Likewise, the ports between the scheduler and the slot enabling the scheduler to "tell" the slot when it executes. This scheme also makes hierarchical scheduling possible and allows other scheduling policies to be specified. The *CPU1* with its task, slot, scheduler, and a formal contract (denoted by "Contract_CPU_Performance") are displayed in Figure 3. The contract assumes that an event on *input_ port* occurs each 100ms and guarantees that the distance between the input event on *input_ port* and the output event on port *ouput_ port* is within 12ms and 15ms.

Note that *CPU1* has only one task receiving data and exporting data back to the bus. This indicates that there might be some bus interface functionality within the task. So maybe it is a good idea to decompose this one task a little further. The following section will describe the lower abstraction level (the component level), in which the technical perspective will be refined.

### 4.3 The Lower Abstraction Level

After designing the initial technical architecture on the upper level, we now proceed to model the technical architecture with more detail on the lower abstraction level. As mentioned above, we need to decompose the air-generation-demand-calculation-application task a bit further into the importer task, the

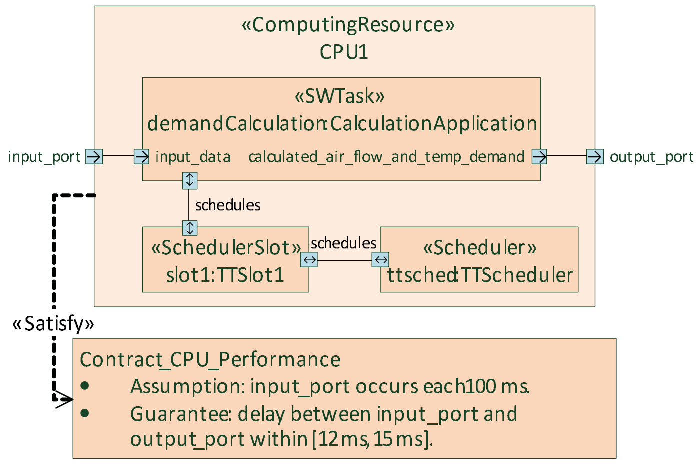

«ComputingResource»
CPU1
«SWTask»
demandCalculation:CalculationApplication
input_port
input_data
calculated_air_flow_and_temp_demand
output_port
schedules
«SchedulerSlot»
slot1:TTSlot1
schedules
«Scheduler»
ttsched:TTScheduler
«Satisfy»
Contract_CPU_Performance
Assumption: input_port occurs each 100 ms.
Guarantee: delay between input_port and output_port within [12 ms, 15 ms].

**Fig. 3.** The CPU1 component with one task.

actual calculation task, and the exporter task. This means, we will refine the computing resource *CPU1* from the upper level.

For the environment part we modeled the type with the two corresponding ports of the CPU. For now, there is no behavior environment model, so there is no further decomposition of the environment component. As for the CPU: It was decomposed into its tasks, slots and the scheduler. Figure 4 depicts the model.

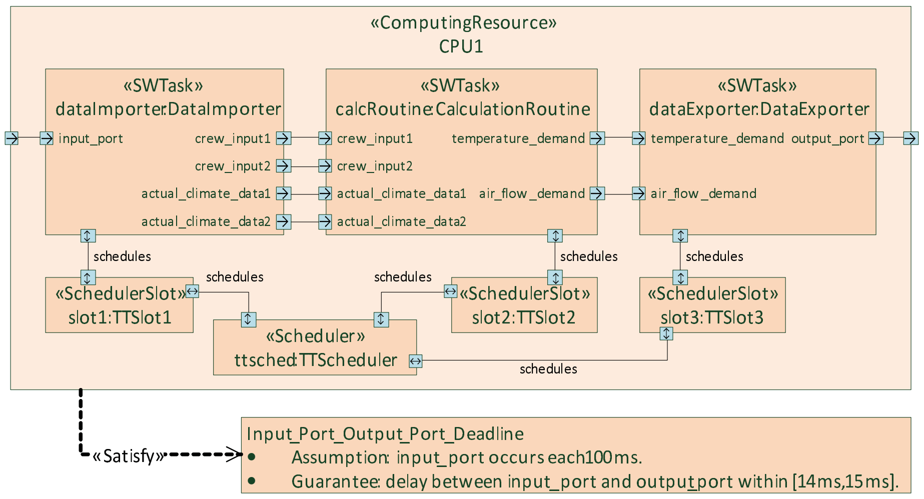


**Fig. 4.** The lower abstraction level CPU1 component with three tasks.

As depicted there are now three tasks in the computing resource: The *dataImporter*, *calculationRoutine*, and the *dataExporter*. The data importer has one input (the connection to the *Input_ Com* communication resource like on the upper level). Likewise, the data exporter has the output port connecting to the *Out_ Com* communication resource. The calculation routine has an input port for each incoming data and an output port for each outgoing data. It can only be activated if all four incoming data packets are available.

There is also a Contract (denoted by "Input_Port_Output_Port_Deadline") that the *CPU1* satsifies: The refined (from the upper level) deadline for the input and output port. Note that the assumption is unchanged, but the deadline interval has changed now (this often happens when traversing from a more abstract design to a more detailed design).

Having the two technical perspectives, one on the upper and one on the lower level, we now need to specify how they are mapped, i. e. how the upper level is refined into the lower level. Figure 5 shows how the task on the upper level is mapped to the tasks on the lower level. In this case it is rather simple, though the ports have different names their types are the same and it is a simple One-Port-To-One-Port mapping. Having said that, it is not obvious how the two contracts on both levels relate to each other.

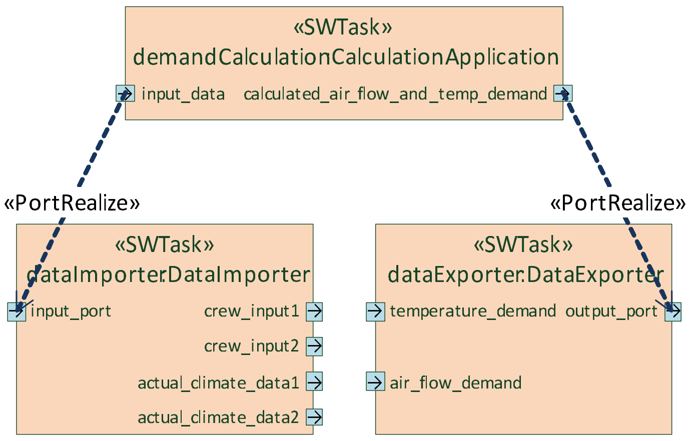


**Fig. 5.** The mapping between the task of the two abstraction levels.

Keep in mind that in the HRC semantics, a connection between two ports describes their equality. So, for delegation connections the contract's parts and ports are the same and for assembly connectors assumption and guarantee have to be checked against each other. In our case this means: The `input_port` of *CPU1* on the upper level is delegated to the task port `input_data` (see Figure 3) denoting their equality. The same is valid for the connection between `output_port` and the `calculated_air_flow_and_temp_demand`. Again, this also holds for the ports on the lower level (see Figure 4). So, with the contracts on the upper and lower level satisfied by the corresponding *CPU1*, parts of these contracts are also valid for the subcomponents of *CPU1*. It so happens that these subcomponents are mapped and thus induce the necessity of an realization test: Is the contract of the coarser component still valid if it is replaced by a finer component? Or in our example: Is the *CPU1* on the lower level a valid realization of the *CPU1* on the upper level, concerning their contracts and mappings? For this we apply the refinement check which formally checks if the answer to the above questions is Yes. The next subsection will describe in detail how this refinement check is performed for our example.

## 4.4 Realization Check between Abstraction Levels

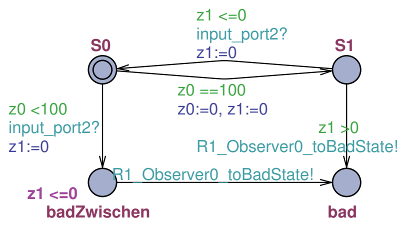
We will apply the technique checking a refinement relation introduced in Section 3.2. For this, we implemented a tool which derives relating contracts out of the system models, parses the RSL pattern with which the contracts are specified, generates a corresponding Uppaal timed automaton network and checks this against the properties illustrated in Section 3.2.

Consider again the contracts of *CPU1* in the upper and lower abstraction level introduced in the previous section. In general we have to check both conditions $[\![A_s]\!] \subseteq \alpha([\![A'_s]\!])$ and $\alpha([\![C']\!]) \subseteq [\![C]\!]$, but as in our example $A_s$ and $A'_s$ are equal, we can directly start checking the second condition. The observer automata resulting from the *CPU1* contract of the upper abstraction level are illustrated in Figure 6. It consists of three parts:

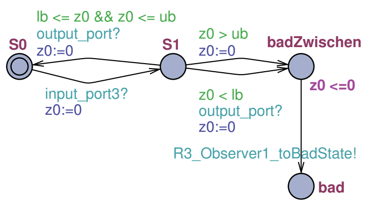
- The automaton $O_A$ in Figure 6(a) results from the assumption of *CPU1* at the upper abstraction level, i. e. $A_s =$ *input_port* occurs each $100ms$. Each $100ms$ it enters its state $S_1$ and expects to receive event `input_port`. If the event is not received timely, it enters its bad state and sends an event `R1_Observer0_toBadState` to the automaton depicted in Figure 6(c).
- The automaton $O_G$ in Figure 6(b) results from the guarantee part of the contract, i. e. $G =$ delay between *input_port* and *output_port* within $[12ms, 15ms]$. It enters its bad state if the delay between event *input_port* and *output_port* is less than $12ms$ or greater than $15ms$.
- The automaton $O_C$ in Figure 6(c) gives the overall state of the contract. If $O_A$ switches to its bad state, the contract is trivially fulfilled and $O_C$ switches to the state `good`. If the $O_G$ switches to its bad state, the $O_A$ must finally switch to its bad state, because otherwise the contract would be falsified.

This observer network is triggered by the automaton network depicted in Figure 7 consisting of two automata:

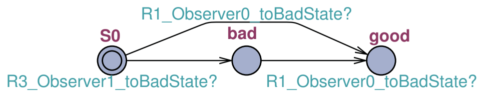
- The automaton in Figure 7(a) results from the assumption of the contract of *CPU1* at the lower abstraction level, i. e. $A_s =$ *input_port* occurs each $100ms$.
- The automaton in Figure 7(b) results from the guarantee part of the contract $G =$ delay between *input_port* and *output_port* within $[14ms, 15ms]$.



(a) Observer automaton for $A_s = input\_port$ occurs each $100ms$.

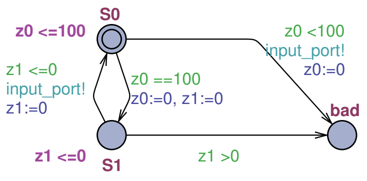


(b) Observer automaton for $G$ = delay between *input_port* and *output_port* within $[12ms, 15ms]$.

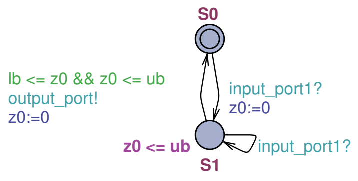


(c) Automata giving the state of the contract.

**Fig. 6.** Automata for the contract of the abstract component.



(a) Trigger automaton for $A_s = input\_port$ occurs each $100ms$.



(b) Trigger automaton for $G$ = delay between *input_port* and *output_port* within $[14ms, 15ms]$.

**Fig. 7.** Automata for the contract of the concrete component.

This automaton network is checked against the property $A\Box$ not $O_C.bad$, where $O_C$ is the observer automaton illustrated in Figure 6(c). This property states that the bad state of $O_C$ is never reached. If the network fulfills this property, we have shown a valid refinement, which is the case in this example.

## 5 Considering Sets of Contracts

In this section we discuss the extension of the refinement check introduced in Section 3 in order to deal with specifications which may consist of a set of contracts. In this section we will omit technical details of the construction of automata networks like e. g. glue automata. Instead, we will focus on the general checking procedure and will give an extended formulation of the proof obligations introduced in Section 3. Further, we will discuss the construction for only a restricted subset of properties since (as we will see in the following subsection) that the generalization is not always applicable. For the assumption parts we will only consider activation patterns stating periodic triggering of a port. The guarantee parts will contain only delay patterns as demonstrated in the example of Section 4.4. More general cases are work in progress.

## 5.1 Extension of the Refinement Check

The specification of a component with more than one contract is obtained by conjugating them. Unfortunately, the conjunction of two or more contracts is not simply the conjunction of all assumptions and all guarantees. Rather there are various possibilities to formulate the conjunction of contracts. We will use the following:

$$C_1 \wedge ... \wedge C_n = \ (\mathcal{A},\ \mathcal{G}),\ \textbf{with}\ \mathcal{A} = \bigwedge_{i=1}^{n} A_i \ \vee \ \bigvee_{i=1}^{n} (A_i \wedge \neg G_i)\ \textbf{and}\ \mathcal{G} = \bigwedge_{i=1}^{n} G_i.$$

The first refinement property $[\![\mathcal{A}]\!] \subseteq [\![\mathcal{A}']\!]$ is checked in two steps (note that in the following we will omit the semantic evaluation brackets for the sake of readability) :

- $\bigwedge_{i=1}^{n} A_i \ \subseteq \ \mathcal{A}'$
- $\bigvee_{i=1}^{n} (A_i \wedge \neg G_i) \ \subseteq \ \mathcal{A}'$

We omit the necessity to construct a timed automaton for the trigger part which generates the union of both parts, by splitting this check into two parts. Note that timed regular languages are not closed under complementation. So this approach does not work in general. For the special case of delay patterns, the complement of the guarantee automata can be easily constructed.

The second refinement condition becomes in the general case $(\mathcal{A}',\ \mathcal{G}') \ \subseteq (\mathcal{A},\ \mathcal{G})$. In the first part of the check we have shown $\mathcal{A} \subseteq \mathcal{A}'$ which is equivalent to $\neg\mathcal{A} \supseteq \neg\mathcal{A}'$. With this we can simplify the left hand side of this term to $\mathcal{G}'$. For technical reasons it could further be necessary to add the assumptions to the guarantee. Consider for example the special case of activation and delay patterns: The assumption part is used to trigger the guarantee part. Without the assumptions an additional trigger structure for the guarantee part would be necessary.

In order to extend the left hand side of the above expression we use the following equivalence (zero set extension):

$$\mathcal{G}' = \ (\bigwedge_{i=1}^{n} A_i' \vee \neg(\bigwedge_{i=1}^{n} A_i')) \wedge \mathcal{G}'.$$

Because it holds that $\neg(\bigwedge_{i=1}^{n} A_i') \wedge \mathcal{G}' \subseteq \mathcal{A}$ we can omit this term. This leads us to the following proof obligation:

$$\bigwedge_{i=1}^{m} (A_i' G_i') \subseteq (\bigvee_{i=1}^{n} \neg A_i \ \wedge \ \bigwedge_{i=1}^{n} (\neg(A_i) \vee G_i) \vee \ \bigwedge_{i=1}^{n} G_i).$$

Again we have the problem of complementing a timed automaton. In the case of the automata $A_p$ resulting from periodic activation patterns we can again find an automaton accepting all words which are not accepted by $A_p$.

### 5.2 Cyclic dependencies

If the specification of a component consists of a set of contracts, we could get cyclic dependencies if an output port is connected to an input port. For this we need a strong causality between events. In [11] this problem is prevented by a stepwise definition of contracts, i. e. all guarantees hold initially and if the assumptions hold up to the $n^{th}$ step then the guarantees hold up to $n+1^{th}$ step. Another way to break such cyclic dependencies is to add delays to the data flow. The cases we considered utilized delay patterns.

## 6 Summary

In this paper we illustrated a technique to verify refinement relations for contract-based specifications for our previously proposed common systems meta-model (CSM). Our CSM allows to structure the design space by concepts like Abstaction Levels and Perspectives. It enables formal specifications via contracts allowing for each aspect to characterize the allowed design context of a component. In order to preserve traceability between model artifacts and to put those into refinement relations, the concept of Mapping was introduced.

In particular, an abstract component which is realized by a more concrete component has a refinement relation to that component. This relation has to respect contract specifications of both components. Thus, we formally defined the refinement relation and introduced a timed automaton based verification technique. For this, we derived a timed automaton network out of the assumption and guarantee parts of relating components and defined necessary properties. Whenever the network adheres to the properties, the refinement relation between the corresponding contracts holds. The resulting network was checked against the properties with the aid of the verification tool Uppaal. For evaluation, our technique was applied to an industrial case study from the avionics domain.

Currently, we are extending our approach of refinement checking in order to deal with more general n-to-m mappings, i. e. where one component is related to a set of more abstract or more detailed components. Especially cycles deserve more research: Interdependent contracts may lead to false conclusions. This problem is widely discussed in literature, e. g. in [11, 14].

In the future, we will analyze for which set of properties our approach is applicable. In this context we will analyze the effects which are occuring when we extend our approach to multi viewpoint analyses.

Further research will be conducted also in the field of deeper analysis methods for more complex mapping functions. In the future we also plan to integrate the refinement check described in this work with a broader evaluation of architecture alternatives in order to guide a developer through a design space exploration process.

## References

1. L. Aceto, A. Burgueño, and K. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054177.
2. A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber. A model-based design methodology with contracts to enhance the development process of safety-critical systems. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'10, pages 59–70, Berlin, Heidelberg, 2010. Springer-Verlag.
3. A. Benveniste, J.-B. Raclet, B. Caillaud, D. Nickovic, R. Passerone, A. Sangiovanni-Vincentelli, T. Henzinger, and K. G. Larsen. Contracts for the design of embedded systems, Part II: Theory. Submitted for publication, 2011.
4. CESAR SP2 Partners. Definition and exemplification of requirements specification language and requirements meta model. CESAR_D_SP2_R2.2_M2_v1.000.pdf on http://www.cesarproject.eu/fileadmin/user_upload/, 2010.
5. W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, march 2011.
6. W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting re-use of embedded automotive applications through rich components. In *Foundations of Interface Technologies, FIT'05*, 2005.
7. A. R. INC. ARINC 653 - Avionics Application Software Standard Interface - Part 1 - Required Services. *Part of ARINC 600-Series Standards for Digital Aircraft & Flight Simulators*, March 2006.
8. B. Josko, Q. Ma, and A. Metzner. Designing Embedded Systems using Heterogeneous Rich Components. *Proceedings of the INCOSE'08*, 2008.
9. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer STTT*, 1(1-2):134–152, 1997.
10. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
11. J. Misra and K. Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, SE-7(4):417 – 426, july 1981.
12. Object Management Group. *OMG Systems Modeling Language (OMG SysML $^{TM}$)*, November 2008. Version 1.1.
13. J. Philipps and B. Rumpe. Refinement of information flow architectures. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 203–, Washington, DC, USA, 1997. IEEE Computer Society.
14. A. Pnueli. *In transition from global to modular temporal reasoning about programs*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
15. R. P. Pontes, M. Essado, P. C. Véras, A. M. Ambrósio, and E. Villani. Model-based refinement of requirement specification: A comparison of two v&v approaches. *ABCM Symposium Series in Mechatronics*, 4(IV.05):374–383, 2010.
16. Project SPEEDS: WP.2.1 Partners. SPEEDS Meta-model Behavioural Semantics — Complement do D.2.1.c. Technical report, The SPEEDS consortium, 2007.
17. The SPEEDS Consortium. SPEEDS Project. `http://www.speeds.eu.com`.

# Model-based Consistency Checks of Electric and Electronic Architectures against Requirements

Nico Adler[1], Philipp Graf[1], and Klaus D. Müller-Glaser[2]

[1] FZI Research Center for Information Technology, Karlsruhe, Germany
{adler,graf}@fzi.de
[2] Institute for Information Processing Technology, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
klaus.mueller-glaser@kit.edu

**Abstract.** The electric and electronic architecture (EEA), which is built up during the concept phase of automotive electronics development, has fundamental impact on the success of a vehicle under development. The complexity of upcoming architectures requires novel approaches to support system architects during the design phase.
This paper describes a model-based generic approach which allows verifying an EEA with regard to its requirements by using techniques of consistency checks during an early design phase. This includes handling of incomplete models. In this case it offers the possibility to automate consistency checks and in future work facilitate an automatism for optimization and design space exploration to check different realization alternatives of an initial EEA. Automatic report generation of results serves for documentation.

**Keywords:** electric and electronic architectures, model-based engineering, automotive, verification, requirements

## 1 Introduction

Electric and electronic architectures (EEA) in the automotive and avionic domain build a complex network of a multitude of embedded systems. In the automotive domain we already see an amount of up to 70 networked electronic control units (ECUs) in current upper class vehicles [1, 2]. Various innovations, e.g. driver assistance systems, and new technologies, e.g. FlexRay, make the EEA of vehicles more complex, including numerous technical and functional aspects. Driven by customer demands for more safety, comfort and infotainment, this bears growing challenges for upcoming development activities [3].

The Original Equipment Manufacturer (OEM) must find new ways to control and manage the rising complexity of an EEA. In addition different variants of the EEA, as a single product can have several equipment concepts, increase the difficulty to analyze, whether an EEA meets all requirements [4, 5]. Moreover new standards like ISO26262 [6] increase quality requirements and efforts to verify an EEA regarding functional safety.

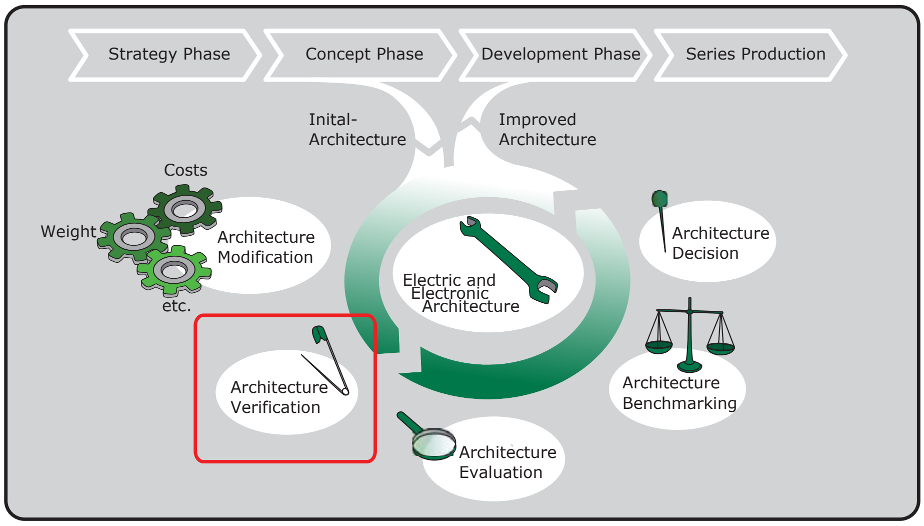


**Fig. 1.** Simplified product life cycle and EEA development cycle [7]

Within the product life cycle the *concept phase* deals with functionality, convenience, risk and profitability of the vehicle [8]. The result must be a concept that meets the main requirements of a vehicle. During the *development phase* the OEM implements the variants of the solution from the *concept phase* into prototypes. This deals with the technical feasibility of alternative solutions. Further phases follow and are out of the scope of this work.

During the *concept phase* about 80 percent of lifecycle costs will be determined, although the *concept phase* itself is only about 6 percent of the total incurred costs [9, 10]. After finalization of the *concept phase* subsequent changes in the EEA are either associated with enormous costs or may not be feasible anymore. Inconsistencies, which cannot be eliminated, in the worst can put the project's success in risk. Therefore, it is a necessity to perform optimizations and verification already during the *concept phase*.

A simplified iterative process for developing an EEA can be described using a cycle with five steps, as shown in Fig.1. First of all, during *architecture modification*, system architects try to improve an existing initial EEA concerning different criteria. This can be done by optimization or by design space exploration. In the second step, the *architecture verification*, they must prove if the modified architecture meets all requirements. This is a very challenging and time-consuming task and can be accomplished using consistency checks. Subsequently the EEA has to be evaluated. This can be done using a cost breakdown structure to get the total system costs with regard to product lifecycle [11]. To find the most acceptable solution and to achieve an *architecture decision*, different realization alternatives are benchmarked.

The development process for vehicle architectures is usually spread over several departments. This results in difficulties for global design decisions and especially in the proof of overall consistency of an EEA. The objective is to verify and demonstrate in an early stage, which (sub-) areas present or could present inconsistencies against requirements, so that measure can be taken to solve them.

This paper focuses on a concept for automatically verifying an EEA during the concept phase regarding requirements and specifications and is organized as follows: The next section briefly defines verification and gives an overview of model-based domain specific languages for EEA. Section 3 briefly relates verification to model-based engineering. The following two sections present our methodology for verification of model-based EEA. A discussion of the adaption to a domain specific toolset for a first prototype is given in Section 6. The final section gives a conclusion and presents future work.

## 2 Related Work

Verification of an EEA is one of the significant points during development. Boehm described the basic objectives of verification and validation (V&V) early in the product life cycle with identification and solving of problems and high risk issues [12]. The V-Model-XT V1.3 describes: ‘verification is to ensure that selected work results meet their requirements’. Therefore a definition of verification procedures and setting up the necessary verification environment must be done [13].

Electric and electronic architecture modeling supports system architects as models abstract complex problems. Different approaches and projects for the model-based description of especially automotive EEA exist, e.g. the project Titus [14], the language EAST-ADL [15], which emerged from the project EAST-EEA, and the EAST-ADL2 [16], which emerged from the follow-up project, and AUTOSAR [17].

Another model-based approach for the description of EEA is the ‘Electric Electronic Architecture - Analysis Design Language’ (EEA-ADL) [18, 19]. This data model also forms the basis of the architecture modeling and analysis tool PREEvision [4]. It combines the previously presented approaches within a tool and was used for the following described implementation. PREEvision Version 3.1 provides seven abstraction layers. *Requirements* and *Feature-Functionality-Network* constitute the first abstraction layer. Artifacts of this layer are text elements, which represent atomic features or requirements to the architecture. The underlying layers are: *logical architecture*, *function network*, *component architecture with network topology*, *electric circuit*, *wiring harness* and the *geometrical topology*. Cross layer links between model artifacts can be modeled using mappings. Apart from modeling EE relevant content, the EEA-ADL provides the opportunity to deposit attributes for costs, weight etc. This makes the model suitable to apply metrics and perform architecture evaluations. For analysis of an EEA an integrated, graphically notated metric framework can be used [20]. To perform consistency checks an integrated consistency rule model editor is given.

Rules for consistency checks can be modeled in a graphical way. Simple rules for checks can be set up easily, but for complex ones with lots of constraints, inputs and multiple involved abstraction layers, this approach is too limited. Another disadvantage of the provided rule modeling is that only the boolean logic operator AND can be used. Therefore, the validness of multiple solutions is hard to model. Consistency checks have to be generated within the consistency rule model and afterwards synchronized with the architecture model. This is very time-consuming, especially for building up and testing complex rules. The created consistency checks can be started only manually. Therefore it is not straightforward to use this methodology for a desired semi-automatism for optimizing an EEA as the checks must be started individually depending on the actual optimization state.

Some of the presented domain specific languages offer an import and export of requirements from requirements management tools like IBM Rational DOORS[3]. The combination of these tools offers to map requirements within the model-based domain specific language to the corresponding artifacts and consistency checks.

## 3 Checking EEA-Consistency in the Context of Model-Based Development

The quality of verification for EEA models is only as good as requirements are described, the according consistency checks are derived from these and realized as executable rules. To do this, it is mandatory that a consistent model exists, which ensures that all EE relevant data is available and up-to-date, including requirements. During the concept phase with consideration of model-based engineering, models are not complete and different sub-parts exist at different detail levels. Therefore model-based verification of constraints has as an additional requirement to secure that verification rules also work with incomplete models. Also, incomplete requirements and specifications either from OEM or supplier have to be taken into account. However absolute maximum ratings of datasheets or specifications can be precalculated or estimated from previous series to perform first consistency checks.

Inconsistency at any point shall not abort verifying as any information about existing consistencies and inconsistencies are beneficial. With automated reporting the results must be captured so that during the development process different versions can be traced and reproduced. Guidelines or regulations of standards must be used as a basis for documentation templates. This is, for example, requested by ISO26262.

It has to be ensured that consistency checks do not apply changes to the EEA data model. The deposited rules are only allowed to retrieve data out of an immutable model.

[3] http://www.ibm.com/software/awdtools/doors, 2011.

## 4 Overview of the Constraint Verification Approach

Starting point for model-based verification is a EEA data model, which is filled with any available and relevant information, as shown in Fig.2 on the left hand side. This can be done using a model-based domain specific language such as the examples presented in Section 2.

Within this model the EEA is specified and requirements must be created or imported from other tools. The requirements linked to the EEA data model are the corner stones for the verification. For the requirements layer textual described and hierarchical constructed requirements are suggested. To structure requirements, requirements packets should be used. Layer internal mappings are used for building up a network between requirements and also requirements packets. In further steps requirements can be mapped to the corresponding consistency checks.

For automatic verification, five different functional blocks, as shown in Fig.2 on the right side, are provided and are described in the following sub-sections: *consistency check blocks* with the checking rules for verification, *model query blocks* concerning data acquisition, *control unit*, *requirement block* and *report generation block* for documentation.

### 4.1 Consistency Check Block

From each requirement or requirements packet at least one consistency check must be derived and implemented as an executable rule. As a decision criterion, the complexity to check the requirement or requirements packet can be used. Therefore the architect has to analyze all requirements separately.

Verification can relate to different abstraction layers in an EEA model. For consistency checks, the corresponding artifacts, including their attributes and their mappings between different abstraction layers, are required. This input data for the consistency check can be provided by outsourced data acquisition using model queries and is described in the following subsection.

The multitude of incoming model data from the model queries has to be preprocessed. This includes sorting and filtering data, furthermore structuring of model data which belongs together. Another task is to capture the required corresponding attributes of model artifacts.

The crucial point to verify the EEA is the execution of the consistency check. This is composed of a sequence and examination rule. In a first step the sequence is partitioned into *consistency checks that cannot be performed* and *consistency checks that can be performed*. It is advisable to inspect if all relevant data for performing the check are available. If any relevant information is not available, the consistency check cannot be performed.

At this point results of not possible checks must be collected and stored for postprocessing and preparation for reporting. If the consistency check can be performed, the examination rule describes what shall be checked and in fact represents the derived requirement or requirements packet. Instructions for checking

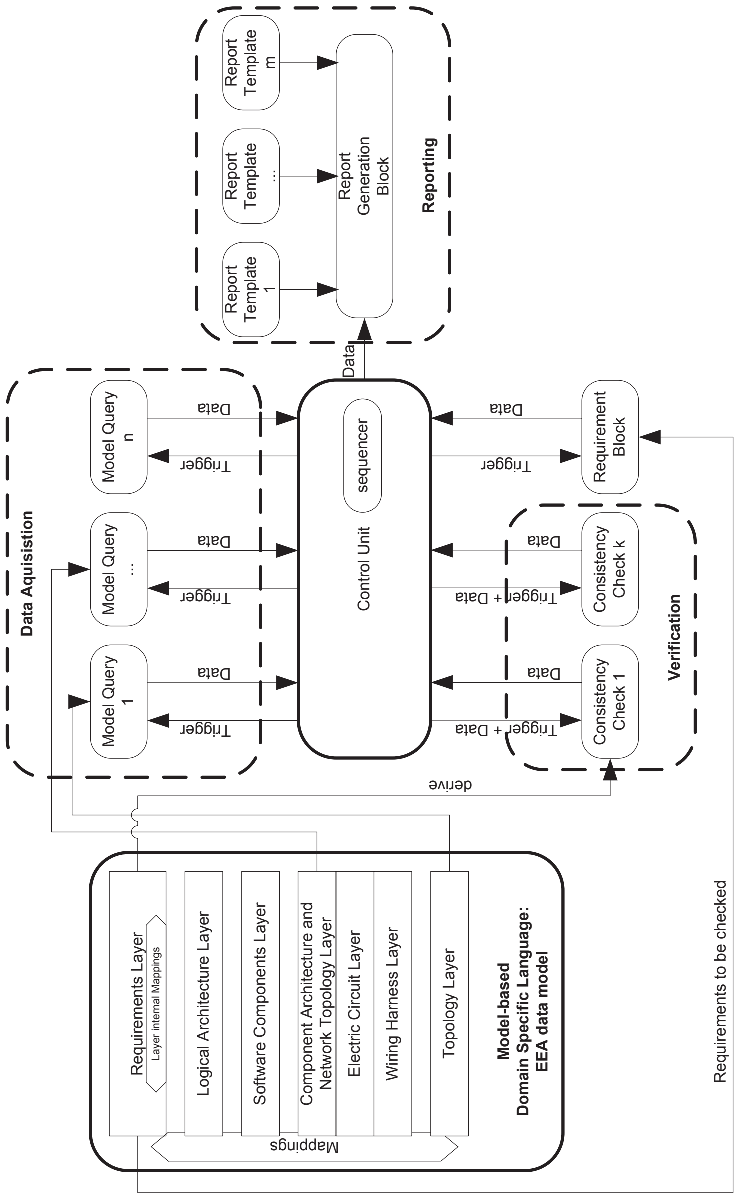


**Fig. 2.** Approach for verification of EEA against requirements

on inconsistency or consistency can be implemented using a specific algorithm. The algorithm examines the available data and executes the rule.

Results of the consistency check are lists or tables. Additionally the degree of a possible violation of a rule can be estimated. Subsequent post-processing is used mainly for preparing the data in a structured way for reporting.

## 4.2 Model Queries Block

Expert knowledge demonstrates that large parts of verification consists of data acquisition and structuring. Model queries can be used for data acquisition and should be based on the corresponding EEA meta-model. This ensures that model queries are correct.

Different consistency checks may require the same model artifacts. The reuse of existing model queries can significantly reduce the overhead of setting up new checks. Hereby model queries can be used for different consistency checks which need equal model artifacts, or the artifacts are part of the model query result and must be extracted by the consistency check block. Therefore, an approach has to be found to avoid redundancy so that model queries are implemented only once. To provide results of model queries to different consistency checks, the model queries must be connected to a *control unit*. The *control unit* forwards the model query results to the corresponding consistency checks.

## 4.3 Control Unit

The *control unit* is the central block. It acts as a sequencer and is connected to all other blocks by input and/or output ports. The ports are used to transmit different kind of data. Two modes are differentiated: *trigger mode* is used for data request and *data mode* is used for transmission of data consisting of header and payload. For both port modes, it is possible to directly connect an input port to one or more output ports. Also incoming triggers or data can be split and/or combined.

Data preprocessing for paths coming from model queries to consistency checks is not provided, because preprocessing can distort results with possible data losses. To avoid this, preprocessing of model query results has to be implemented directly in consistency checks. However data preprocessing is applied for the path coming from consistency check output ports to report generation. Therefore the *control unit* is further connected with the report engine. After final preparation of verification results, the *control unit* transfers data together with report filename to the report generation block.

The subsequent execution of the verification is to be started by the *control unit*. Thus the *control unit* needs information about which requirements should be verified. For this, a *requirements block* as a further block is incorporated.

## 4.4 Requirements Block

The *requirements block* is used to insert a collection of the demanded requirements which have to be verified. Therefore the corresponding requirements from the *requirements layer* are mapped to the block. As data inputs, the *control unit* retrieves them and triggers the derived consistency checks. No mapping means that all consistency checks must be executed.

### 4.5 Report Generation Block

Reporting for documentation is an important tasks to reproduce and capture results. Reporting can be differentiated into documentation for internal or external use. Therefore, an individual configurable template based approach must be provided, as in different departments they have to fill in different forms or bring a different proof respective to standards or recommended practice.

For internal use, there also can be some kind of documentation, but the key factor is to design an EEA which meets all the requirements as soon as possible. Therefore additional identified information can be documented.

The *control unit* must decide at runtime, which results have to be sent to the *report generation block* to fill the corresponding placeholders in the templates. Result export to other tools must also be performable. For this purpose e.g. a XML Schema Definition (XSD) can be implemented using the templates. This offers a wide range to interface other tools for further processing of results. Therefore the generation of documents shall be delivered through a suitable report engine, which can access the prepared templates.

## 5 Execution of Verification

The typical sequence for a model-based verification is shown in Fig.3. In the vertical ‘swimming lanes’ (columns) the five different blocks are presented. Starting point is the *control unit*. The *control unit* requests the *requirement block* for information which requirements should be checked by using a trigger. The bundled requirements are sent to the control unit which selects the model queries to be executed for the corresponding consistency checks. The results in the form of lists or tables are sent back to the *control unit*. The model queries are executed only once, because during verification the EEA data model is not modified. At this point a loop starts. For every requested verification, the *control unit block* bundles the demanded model queries results and sends them to the corresponding *consistency check block*. The verification sequence starts and therefore the examination. Relevant results of the consistency check are sent to the *control unit*, which forwards them after potential preparation for reporting to the *report generation block*.

## 6 Prototype Implementation in PREEvision

Integrated consistency checks require an integrated model as described in Section 2. PREEvision is a software tool that allows persistent modeling and evaluation from requirements down to topology and was used for the first prototyping the concept described before.

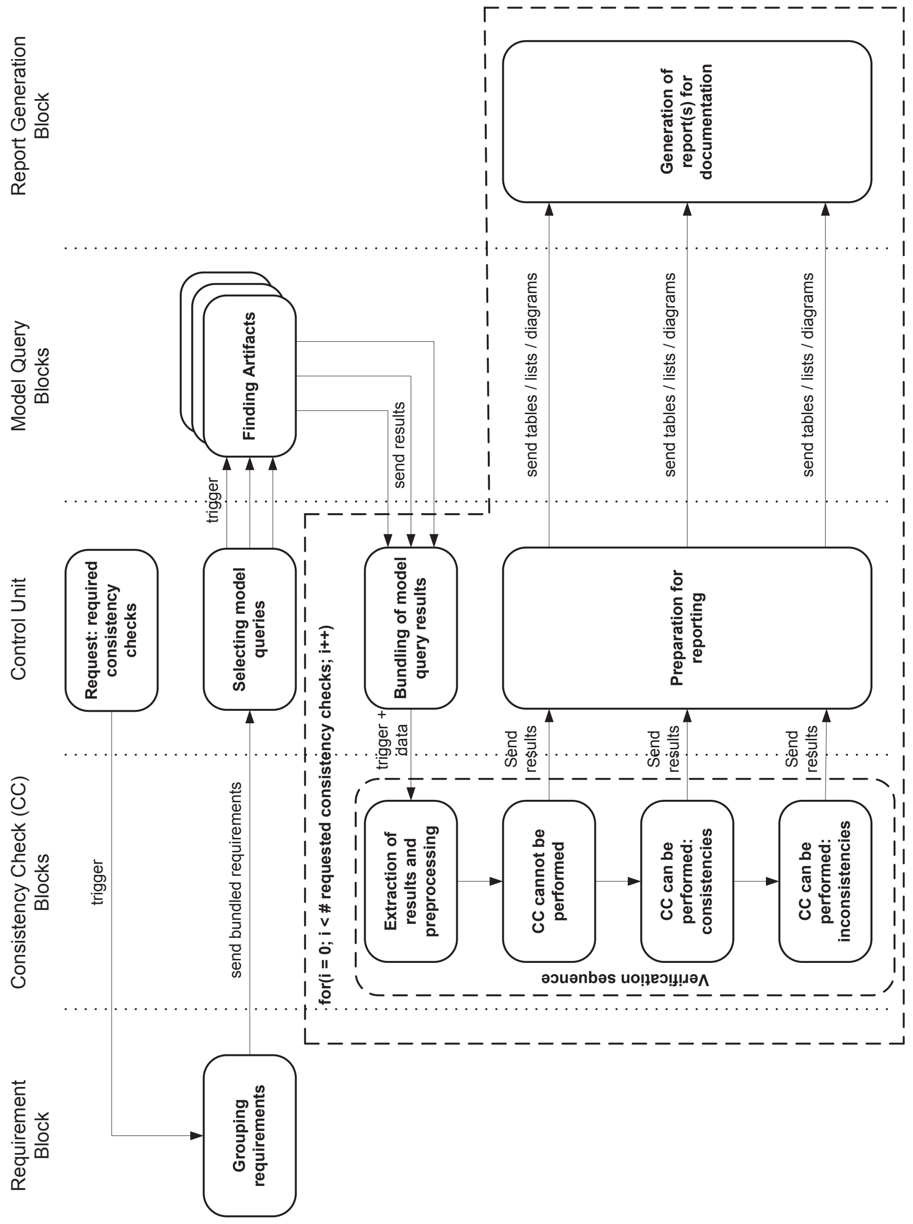

Requirement Block
Consistency Check (CC) Blocks
Control Unit
Model Query Blocks
Report Generation Block
Grouping requirements
Request: required consistency checks
trigger
Selecting model queries
send bundled requirements
trigger
Finding Artifacts
send results
Bundling of model query results
for(i = 0; i < # requested consistency checks; i++)
trigger + data
Extraction of results and preprocessing
CC cannot be performed
CC can be performed: consistencies
CC can be performed: inconsistencies
Verification sequence
Send results
Send results
Send results
Preparation for reporting
send tables / lists / diagrams
send tables / lists / diagrams
send tables / lists / diagrams
Generation of report(s) for documentation

**Fig. 3.** Execution of verification

As a simple example we demonstrate the approach with a consistency check derived from the requirement 'All ECUs with the attribute *isPartOfActiveVariant = true* and *availability in crash* is set to *low, medium* or *high* must be allocated to corresponding *installation spaces*!'

For preparation we modeled an exemplary EEA within PREEvision. This consisted of *component layer* with *ECU* artifacts, the *topology layer* with *InstallationLocation* artifacts and added *HardwareToTopology-Mappings* between the two abstraction layers. The requirement was inserted in the *requirements layer* of the EEA data model.

### 6.1 Model Query Blocks

Afterwards we identified which artifacts from different abstraction layers are relevant for performing a consistency check. In this case we needed the involved *ECU* artifacts from the *component layer* including the attribute *availability in crash.* This formed the first model query. Further we needed all *InstallationLocations.* This formed the second model query. The last model query was to find all existing ECU mappings from the component into the *topology layer.*

For the model queries, the integrated rule model in PREEvision was used which is based on the corresponding EEA data model. It allows to model rules in a graphical way. Complex patterns to match can be defined, using not only the *source-object* and its properties, but also objects and *LinkPairs* between the objects, as shown in Fig.4 on the left hand side for the rule diagram *ECUtoInstallationLocationMappings.* A further restriction was added to the example by using one of the *attributes* of the *ECU*, thus the *isPartOfActiveVariant* was set to boolean value *true.*

We generated the rules for the three required model queries, that can be used within the metric framework. The results after execution of the model queries are tables or lists shown on the right hand side in Fig.4. These can be further processed by the consistency checks.

### 6.2 Consistency Check Blocks

For the *consistency check block* a Java-based calculation block within the metric editor was used. With this approach we access EEA model artifacts and their attributes using Java as a programming language instead of the graphical rule modeling as for the model queries. Being more flexible, it allows to construct simple to very complex consistency checks. The results of the *model query blocks* are used as their input.

It is also possible to construct hierarchical consistency checks using several calculation blocks. For example this can be used to subdivide a requirement or requirements packet into several consistency checks. In this case trigger and data paths have to be looped through the parent *consistency check blocks.*

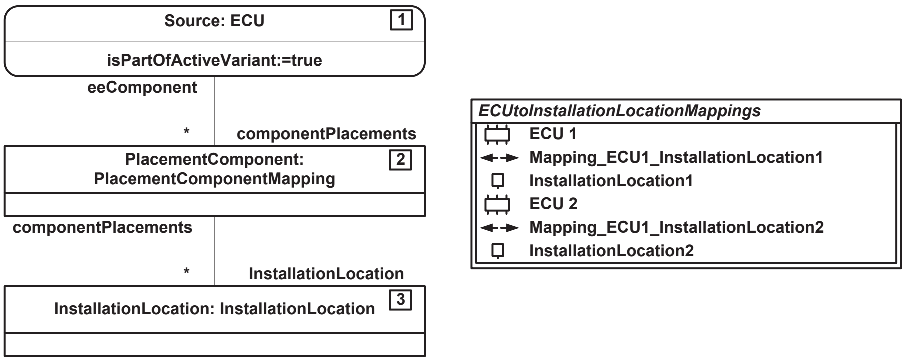

Source: ECU
1
isPartOfActiveVariant:=true
eeComponent
*
componentPlacements
PlacementComponent: PlacementComponentMapping
2
componentPlacements
*
InstallationLocation
InstallationLocation: InstallationLocation
3
ECUtoInstallationLocationMappings
ECU 1
Mapping_ECU1_InstallationLocation1
InstallationLocation1
ECU 2
Mapping_ECU1_InstallationLocation2
InstallationLocation2

**Fig. 4.** Rule diagram for the model query (left) and results (right)

### 6.3 Control Unit

The *control unit* was implemented using a calculation block and was connected with all other blocks using data flows. It contains the allocation table of model queries to the corresponding consistency checks. The execution sequence was implemented and internal trigger and data paths were connected. It is possible to individually improve existing consistency checks or to add new ones. Extension of any kind can be performed easily, as existing model queries, etc. can be reused and only the allocation table in the control unit has to be updated.

### 6.4 Report Generation Block

For the *report generation block*, we used the open source templating *Apache Velocity Engine*[4], which is integrated in the PREEvision metric framework. Velocity permits to use a simple template language to reference objects defined in Java code. As the output format for the first prototype we chose HTML for the generated files. This allows graphical layout of results and ensures traceability using hyperlinks. Using velocity templates we formed the basic structure of the graphical appearance including a navigation bar. Placeholders in the velocity templates were filled with the data coming from *consistency check blocks*, looped through the *control unit*. To obtain better overview of the identified inconsistencies we export the corresponding diagrams to PNG-file format automatically and include them in the reports.

## 7 Conclusion and Future Work

In this paper we have presented an approach that makes it possible to automate verification for electric and electronic architectures already during concept phase

[4] http://velocity.apache.org, July 2011

using consistency checks in a model-based way. The methodology is generic and even incomplete models can be checked. This is a significant step to support the system architect concerning reduction of development time and ensures EEA being consistent against requirements. Integrated reporting serves for documentation.

The developed methodology and its implementation in PREEvision has shown to work in our first prototype. An analysis and application of the approach in EAST-ADL is being considered. Also, adapting to an existing standard for expressing constraints, the Object Constraint Language (OCL) which is included in the Unified Modeling Language (UML), will be analysed.

Future work will mainly focus on expanding the approach to architecture evaluation using metrics for calculating quality of the EEA. This ability can be used for benchmarking different EEA realization alternatives. In further steps, the approach can be extended to (semi-) automatic optimization and design space exploration. For generating new EEA realization alternatives a strategy for design space exploration must be found. For this purpose, the automatic verification can deliver useful information about the degree of compliance to requirements for a new generated EEA realization alternative.

Application and evaluation of the approach with a real-world EEA model is planed for a case study, but will require the cooperation with an Original Equipment Manufacturer (OEM) to bring in the real-word application as its intellectual property.

**Acknowledgements.** This document is based on the SAFE project in the framework of the ITEA2, EUREKA cluster programme 3674. The work has been funded by the German Ministry for Education and Research (BMBF) under the funding ID 01IS11019. The responsibility for the content rests with the authors.

## References

[1] Larses, O.: Architecting and Modeling Automotive Embedded Systems. doctoral dissertation. Stockholm (2005)

[2] Reichart, G., Haneberg, M.: Key Drivers for a Future System Architecture in Vehicles. SAE Convergence 2004, Vehicle Electronics to Digital Mobility. Detroit (2004)

[3] Hillenbrand, M., Heinz, M., Adler, N., Müller-Glaser, K.D., Matheis, J., Reichmann, C.: ISO/DIS 26262 in the Context of Electric and Electronic Architecture Modeling. Architecting Critical Systems ISARCS (2010)

[4] aquintos GmbH: PREEvision Version 3.1 Manual, www.aquintos.com. Karlsruhe, Germany (2010)

[5] Burgdorf, F.: Eine kunden- und lebenszyklusorientierte Produktfamilienabsicherung für die Automobilindustrie. doctoral dissertation. Karlsruhe Institute of Technology, Karlsruhe (2010)

[6] International Organization for Standardization: ISO/DIS 26262 Roadvehicles-Functional Safety. Part 1 - 10, www.iso.org, Tech. Rep. (2010)

[7] Adler, N., Gebauer, D., Reichmann, C., Müller-Glaser, K.D.: Modellbasierte Erfassung von Optimierungsaktivitäten als Grundlage zur Systemoptimierung von Elektrik-/Elektronik-Architekturen. 14. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV) 2011, OFFIS - Institut für Informatik, Oldenburg (2011)

[8] Kuster, J., Huber, E., Lippmann, R., Schmid, A., Schneider, E., Witschi, U., Wüst, R.: Handbuch Projektmanagement. Springer (2008)

[9] Voigt, K.I.: Industrielles Management - Industriebetriebslehre aus prozessorientierter Sicht. Springer-Verlag Berlin Heidelberg (2008)

[10] Bürgel, H.D., Zeller, A.: Controlling kritischer Erfolgsfaktoren in Forschung und Entwicklung. Controlling, Vol. 4 (1997), Nr. 9, S. 218-225

[11] Blanchard, B. S., Fabrycky, W. J.: Systems Engineering and Analysis. Pearson Prentice Hall, New Jersey (2006)

[12] Boehm, B.: Verifying and Validating Software Requirements and Design Specifications. IEEE Softw., Los Alamitos, CA, USA (1984)

[13] V-Modell-XT Version 1.3, Part 7: V-Modell Reference Mapping to Standards. (2009)

[14] Eisenmann, J., Köhn, M., Lanches, P., Müller, A.: Entwurf und Implementierung von Fahrzeugsteuerungsfunktionen auf Basis der TITUS Client/Serverarchitektur. VDI Berichte, Nr. 1374, VDI-Gesellschaft Fahrzeug- und Verkehrssicherheit, Systemengineering in der Kfz-Entwicklung, (1997)

[15] The East-EEA Project: Definition of language for automotive embedded electronic architecture approach. Technical Report, ITEA, Deliverable D3.6, (2004)

[16] The ATESST Consortium: EAST ADL 2.0 specification. Technischer Bericht, ITEA, (2007), http://www.atesst.org/

[17] AUTOSAR, Automotive Open System Architecture. http://www.autosar.org, (2010)

[18] Matheis, J.: Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262. doctoral dissertation. Karlsruhe Institute of Technology, Karlsruhe (2010), ISBN: 978-3-8322-8968-3

[19] Belschner, R., Freess, J., Mroko, M.: Gesamtheitlicher Entwicklungsansatz für Entwurf, Dokumentation und Bewertung von E/E Architekturen. VDI Bericht, Nr. 1907, S. 511-521, VDI-Verlag, Düsseldorf (2005).

[20] Gebauer, D., Matheis, J., Kühl, M., Müller-Glaser, K.D,: Integrierter, graphisch notierter Ansatz zur Bewertung von Elektrik/Elektronik- Architekturen im Fahrzeug. HDT (Haus der Technik), (2009)

# Modeling and Prototyping of Real-Time Embedded Software Architectural Designs with Colored Petri Nets

Robert G. Pettit IV[1], Hassan Gomaa[2], and Julie S. Fant[1]

[1] The Aerospace Corporation,
Chantilly, Virginia, USA
{robert.g.pettit, julie.s.fant}@aero.org
[2] George Mason University
Fairfax, Virginia, USA
{hgomaa}@gmu.edu

**Abstract.** This paper describes an approach for constructing rapid prototypes to assess the behavioral characteristics of real-time embedded software architecture designs. Starting with a software architecture design nominally developed the using COMET concurrent object-oriented design method, an executable Colored Petri Net (CPN) prototype of the software architecture is developed. This prototype allows an engineer / analyst to explore behavioral and performance properties of a software architecture design prior to implementation. This approach is suitable both for the engineering team developing the software architecture as well as independent assessors responsible for oversight of the software architecture design.

**Keywords:** UML, rapid prototyping. coloured Petri-nets, real-time, embedded, concurrent, software architecture.

## 1 Introduction

The increasing complexity of software-intensive real-time embedded systems, particularly with respect to the behavior of concurrently executing software tasks, requires a thorough understanding of software architecture behavioral properties and tradeoffs among design decisions. Analyzing and understanding the concurrent behavior of real-time embedded software architectures during the early design stages is imperative to the successful and cost-effective development of the system. To address this issue, we present an approach for constructing rapid prototypes of embedded systems to assess the behavioral characteristics of concurrent software architecture designs. The approach leverages software design nominally developed using the COMET concurrent object-oriented design method [1] and reusable Colored Petri Net (CPN) [2] templates and components to rapidly prototype a concurrent software architecture. The goal of the CPN prototype is to compare and assess concurrent software architecture behavior to determine if the software architecture is feasible before spending valuable resources on hardware purchase, development, testing, etc. This paper expands on previous work [2] by specifically focusing on rapid prototyping / independent analysis of concurrent software architectures using reusable CPN components and templates. The complete set of CPN templates for the Unified Modeling Language (UML) [23] behavioral patterns used in this approach were defined in [2]. The resulting approach should provide the ability to quickly develop prototypes of software architecture.

### 1.1 Related Research

Prototyping the concurrent behavior of a real-time embedded system at design time is important to determine whether the system, with its set of concurrent tasks, behaves as desired both in terms of functionality and performance. If potential problems can be detected early in the life cycle, steps can be taken to overcome them.

Typical modeling and analysis methods include event sequence and queuing modeling [1, 3]; simulation modeling [4]; and scheduling analysis [3, 5, 6]. In recent years, there has been an increased effort to construct executable models of software designs and thus allow the logic of the design to be simulated and tested before the design is implemented. Existing modeling tools such as IBM® Rational® Rose® Technical Developer [7] and Ilogix Rhapsody [8] frequently use statecharts [9] as the key underlying mechanism for dynamic model execution. An alternative approach is to model concurrent object behavior using Petri Nets [10-14]. Our efforts [2, 14] have specifically focused on a Colored Petri Net (CPN) approach in which behavioral patterns are identified for objects via UML stereotypes in the software architecture and then modeled with CPN templates matching the behavioral patterns. We have chosen this approach since CPNs provide excellent modeling, analysis, and simulation capabilities for concurrent systems. Additionally, our approach supports independent assessments of the software architecture without requiring the software architect to adapt to a new paradigm. Furthermore, while our method for constructing architecture and design models is based on the COMET [1] approach, any design method that provides guidance on identification and classification of object roles and the structuring of concurrent tasks would be sufficient for our CPN modeling and analysis approach. With respect to COMET, we specifically use the stereotyped behavioral patterns for class roles, including input/output classes; control classes; entity classes; and algorithmic classes. We also use COMET's strategies for structuring concurrent tasks using UML active objects.

## 2 Rapid Prototyping Approach

The purpose of this paper is to describe an approach leveraging executable CPNs for the rapid prototyping of the behavior of communicating, concurrent tasks that make up the software architecture design of a real-time embedded. The purpose of the CPN prototypes proposed in this approach is to simulate the concurrently executing software tasks and to enable analysis and understanding of the concurrent behavior during the early design stages.

The proposed rapid prototyping approach has four major steps that are: 1) Develop the platform independent software architecture 2) Create the platform specific software architecture 3) Construct the CPN prototype 4) Execute and analyze the CPN prototype. Each step is described below in more detail.

### 2.1 Develop the Platform Independent Software Architecture Model

The first step in our approach is to develop the platform independent software architecture model (PIM). The purpose of the PIM is to capture the concurrent object behavior in the form of concurrent behavioral design patterns (BDP), which in subsequent steps will be mapped to CPN templates or components [2]. As discussed in previous work, each BDP represents the behavior of concurrent objects together with associated message communication constructs, and is depicted on a UML concurrent interaction diagram. Each object is assigned a behavioral role (such as I/O, entity, or control) which is given by the COMET concurrent object structuring criteria [1] and depicted by a UML stereotype. An example of a behavioral design pattern for an asynchronous device input concurrent object is given in Figure 1. Note that these behavioral patterns are commonly seen across the UML community as «bondary», «entity», and «control». With out approach using the COMET method, however, additional details are provided for such things as specifying input and output, identifying concurrency properties, and defining state-dependent behaviors.

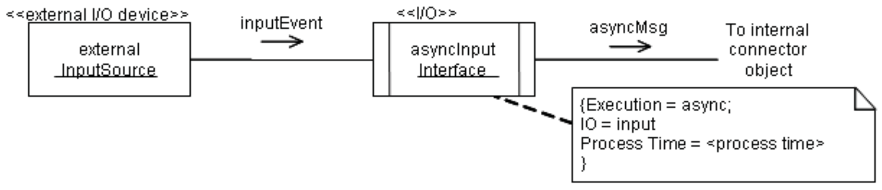


**Fig. 1.** Asynchronous input concurrent object behavioral design pattern

### 2.2 Create the Platform Specific Software Architecture Model

The second step in our rapid prototyping approach is to develop the platform specific software architecture model (PSM). The purpose of the PSM is to capture the performance characteristics of how the software architecture will perform if implemented on a specific platform. To enable fast construction of PSMs, the UML PIM model should be annotated with platform specific characteristics. This is quicker than creating a separate or external PSM model.

Platform specific characteristics and values can then be directly added to the UML software architecture model using a UML Profile such as the UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) [21]. The MARTE Profile provides the ability to capture non-functional performance characteristics directly in UML models. For example, tagging a message in an interaction diagram with the <<paStep>> stereotype indicates that it is a step in sequence that uses resources. The specific platform specific values, such as execution time, can be captured in the stereotype's tags like *hostDemand*.

Performance values can be determined from published information about the platform, as well as through measurement. Note that multiple PSMs can be applied to a given architecture, supporting prototypes for tradeoff analyses. These models may also be constructed at varying levels of fidelity depending on available information. As the development efforts mature, so then can the prototypes of the architecture. During this process, collaboration with domain experts and systems engineers is highly recommended in order to capture the most realistic and complete set of platform specifications. Section 3 of this paper illustrates both a PIM and PSM for a robot controller case study.

## 2.3 Construct the CPN Prototype

After the PSM is developed, an executable CPN prototype from the PSM can be systematically constructed. For each BDP in the PSM (identified by a UML stereotype), a self-contained CPN template is required, which by means of its places, transitions, and tokens, models a given concurrent behavioral pattern. A set of existing reusable CPN templates can be found in [2]. These templates include: I/O (boundary); entity; control; and algorithm. As an example, Figure 2 is the CPN template for an asynchronous device input concurrent object shown in Figure 1.

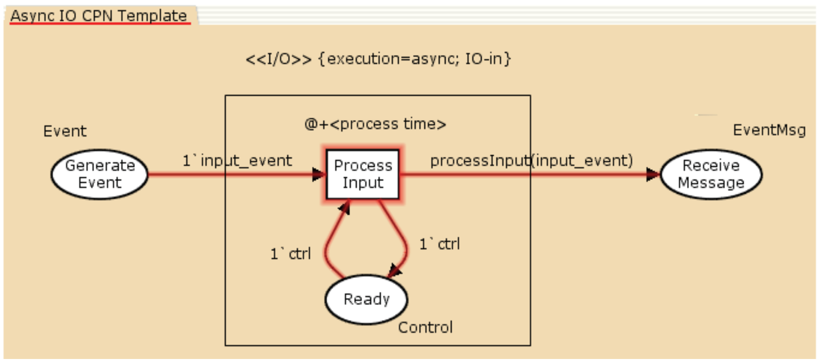


**Fig. 2.** Asynchronous input concurrent object CPN template

To instantiate the templates for each specific object, an analyst using our approach must provide a certain set of architectural parameters captured by following tagged values:

-<u>Execution Type</u>: passive, asynchronous, or periodic
-<u>IO</u>: input, output, or I/O
-<u>Communication Type</u>: synchronous or asynchronous
-<u>Activation Time</u>: periodic activation rate
-<u>Processing Time</u>: estimated execution time for one cycle
-<u>Operation Type</u>: read or write
-<u>Statechart</u>: for each «state dependent» object.

To illustrate pairing these architectural parameters with BDPs, refer to Figures 1 and 2. Figure 1 is an active object, "asyncInputInterface" that implements the I/O behavioral pattern as indicated by its stereotype. Furthermore, tagged types are used to capture specific architectural properties of the object, namely that it executes asynchronously; handles only input; and has a yet-to-be specified processing time of <process time>. The resulting CPN representation in Figure 2 reflects these parameters with the selection of an asynchronous, input-only CPN template and by setting the time inscription on the Process Input transition to @+<process time>.

This <process time> parameter is an estimate of the time required by the object to complete one activation cycle. This information can be obtained directly from UML MARTE annotations in the PSM. For example, process time can be found in the <<paStep>> stereotype in the *hostDemand* tag.

Since CPN templates provide only the basic behavioral pattern and component connections, they must be refined to provide application specific behavior.

To rapidly support construction of the prototype, we recommend using a reuse repository of CPN components. A CPN component is an elaborated CPN template for a commonly used object. For example, if a company commonly uses a specific sensor, a CPN component can be created for the software controller for the particular sensor. This CPN component can then be reused quickly in multiple different prototypes. Reusing CPN components will ultimately reduce the time it takes to construct the CPN prototypes. This is critical in rapid prototyping environments.

After all the BDPs in the PSM have an associated CPN templates or CPN components, the CPN templates and components are then interconnected via connector templates to create a prototype of the software architecture. The CPN prototype is then executed using a CPN tool, thereby allowing the designer to analyze both the concurrent behavior of the CPN prototype, with a given external workload applied to it.

## 3. Case Study: Robot Control

We illustrate our rapid prototyping approach using a robot controller case study based on the Lego® Robotics Invention System™ (RIS), commonly known as Mindstorms™ [16]. The RIS platform was chosen based on the embedded nature of the platform with easily reconfigurable sensors and actuators [18].

The robot controller case study is an autonomous rover employing an infrared light sensor and two motors (actuators). The goal of the rover is to search an area for colored discs, while staying within the course boundary and avoiding obstacles. In this case study, the light sensor is the sole input sensor, responsible for detecting boundary markings, obstacle markings, and discs according to different color schemes. This case study was used as a term project for a graduate course on real-time embedded software engineering at George Mason University.

### 3.1 Robot Controller PIM

The architecture model for the autonomous rover is illustrated in Figure 3. In this particular scenario, we are interested in navigating the course; responding to changes from the light sensor; and taking the appropriate action based on the detection event.

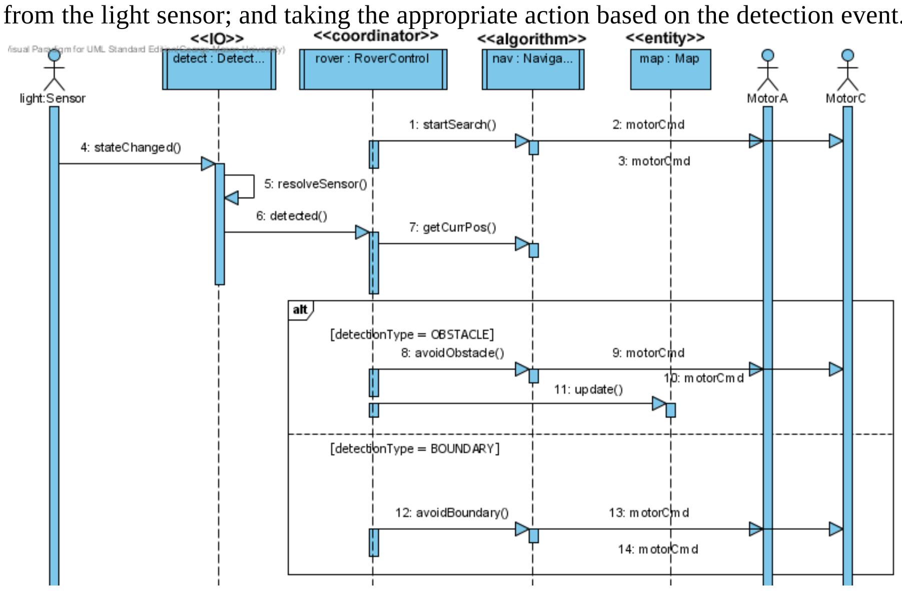


**Fig. 3.** Robot controller PIM interaction diagram

In this design, there are three active, concurrently executing objects (detect, rover, and nav) and one passive object (map). External I/O objects (depicted as actors in Figure 3) are also shown for receiving light sensor input and for modeling output to the two motors. Following object structuring guidelines from the COMET method, each of the objects in the system is stereotyped according to the hierarchy previously shown in Figure 1. These stereotypes indicate the behavioral design pattern (BDP) implemented by each object. Further details about the behavioral properties are augmented with the architectural parameters as follows:

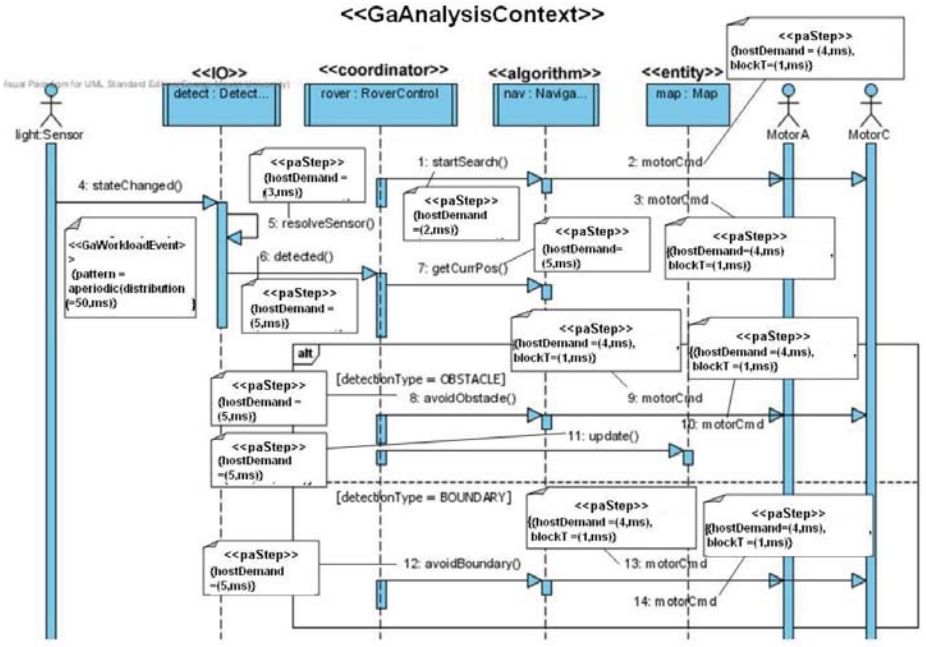
The detect, rover, and nav objects all operate asynchronously and have an Execution Type tagged value of "async". As the input interface for the light sensor, the detect object has an IO tagged value of "input". All messages between the active objects have a Communication Type tagged value of "synchronous", indicating synchronous, buffered communication. This particular design decision was made to decrease the risk of missing a boundary or obstacle detection event. Other design choices for this system would be to employ FIFO or priority queuing. The affects of these design decisions could also be analyzed using the techniques presented in this paper, but are not shown due to space limitations. Finally, the update() operation on the map object has an Operation Type tagged value of "writer".

Note that values for the Processing Time parameters are left unspecified at this point as we will set these parameters based on the PSM in the next section.

### 3.2 Robot Controller PSM

The next step in our approach is to create a platform specific model for the target platform. This is shown in Figure 4 using historical data and hardware specifications for the RIS system [18-20].

In this model, our rover is identified as the single node in the system and is based on the Robot Command eXplorer (RCX) platform. The RCX is the central component to any RIS system and houses the Hitachi H8 microcontroller with a 16 MHz CPU. Paired with the leJOS Java environment the execution speed of this CPU is documented to be 1750 operations per second (OPS).



<<GaAnalysisContext>>
<<IO>>
<<coordinator>>
<<algorithm>>
<<entity>>
detect : Detect...
rover : RoverControl
nav : Naviga...
map : Map
light:Sensor
MotorA
MotorC
<<paStep>> {hostDemand = (4,ms), blockT=(1,ms)}
1: startSearch()
2: motorCmd
<<paStep>> {hostDemand = (3,ms)}
4: stateChanged()
<<paStep>> {hostDemand =(2,ms)}
3: motorCmd
5: resolveSensor()
<<GaWorkloadEvent>> {pattern = aperiodic(distribution =50,ms)}
6: detected()
<<paStep>> {hostDemand= (5,ms)}
7: getCurrPos()
<<paStep>> {hostDemand=(4,ms) blockT=(1,ms)}
<<paStep>> {hostDemand = (5,ms)}
alt
<<paStep>> {hostDemand =(4,ms), blockT=(1,ms)}
<<paStep>> {hostDemand =(4,ms), blockT =(1,ms)}
[detectionType = OBSTACLE]
<<paStep>> {hostDemand = (5,ms)}
8: avoidObstacle()
9: motorCmd
10: motorCmd
<<paStep>> {hostDemand =(5,ms)}
11: update()
[detectionType = BOUNDARY]
<<paStep>> {hostDemand =(4,ms), blockT =(1,ms)}
<<paStep>> {hostDemand=(4,ms), blockT =(1,ms)}
<<paStep>> {hostDemand =(5,ms)}
12: avoidBoundary()
13: motorCmd
14: motorCmd

**Fig. 4.** Robot controller PSM interaction diagram

Additionally, there are 16 KB of ROM and 28 KB of RAM available on the RCX of which, 17.5 KB of RAM are used by the leJOS operating system. The system clock resolution on the RCX, at 1ms, is longer than the time required for observed context switching between concurrent threads, thus the leJOS.overhead is set to zero. In our system, there are three physical devices attached: one light sensor at port S2 and two motors at ports A and C. Independent control of these motors is used to steer the rover; turning is achieved by rotating the left (Motor A) and right (Motor C) motors in opposite directions. Using historical data, the detection latency of the light sensor was set at 10.3 ms, while the output latency of the motors was set at 1 ms.

It would also be useful to combine the information from the PSM with historical data on software size. This type of information is commonly maintained by software development organizations and, in our case, we will rely on average software sizes across the set of student projects. From this data, we discover the following:

«IO» objects average 19 instructions (in Java bytecode) per execution cycle and have an average size of 1,182 bytes.

«coordinator» objects average 27 instructions and 2,722 bytes.

«algorithm» objects average 89 instructions and 1,015 bytes per algorithm.

«entity» objects average 1,400 bytes.

Now, using the combined historical sizing data and information from the rover, we can augment the PSM with this platform specific information. Prior to the CPU being available for performance measurement, an initial estimate of the execution time is computed by multiplying the estimated average number of instructions by the computational speed of the CPU (1750 OPS in our PSM example). These estimates are captured in the hostDemand tag.

### 3.3 CPN Prototype

Using the above PSM design information, we can now begin to construct a Colored Petri Net (CPN) prototype of the software architecture [2]. Using our approach, we start with a context level model, capturing the system as a black box (transition) and external sensors and actuators represented as places. This model, allowing us to focus on the highest level of abstraction with observed inputs and outputs is shown in Figure 5.

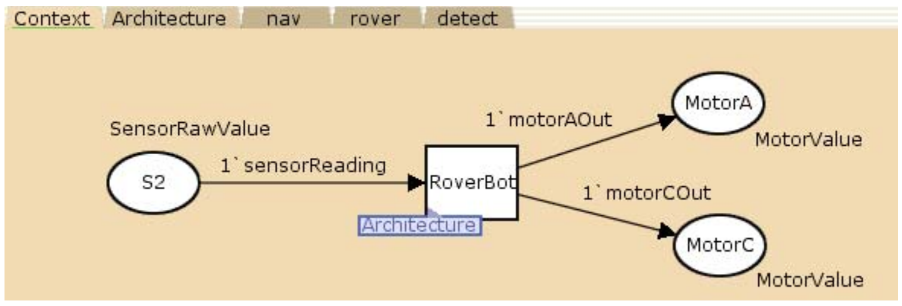


**Fig. 5.** Robot controller context level CPN

Moving forward, our second step is to decompose the RoverBot system-level transition into a layer of abstraction representing the concurrent object architecture. This architecture level model is shown in Figure 6. At this level, each of the active objects from is represented as its own transition (box) in the CPN prototype. Each of these will be further decomposed to implement the specific CPN template matching the objects behavioral design pattern or a CPN component if one exists for the object. We have also included the single «entity» object containing map data and it is represented by a place for the map data to be stored along with a transition and two places representing the behavior for calling the update() operation. Finally, as all message communication between active objects in the RoverBot system is synchronous, there is a CPN place modeling a buffer for the synchronous communication between detect and rover and between rover and nav. Notice that our external input and output places have also been carried down to this level as well.

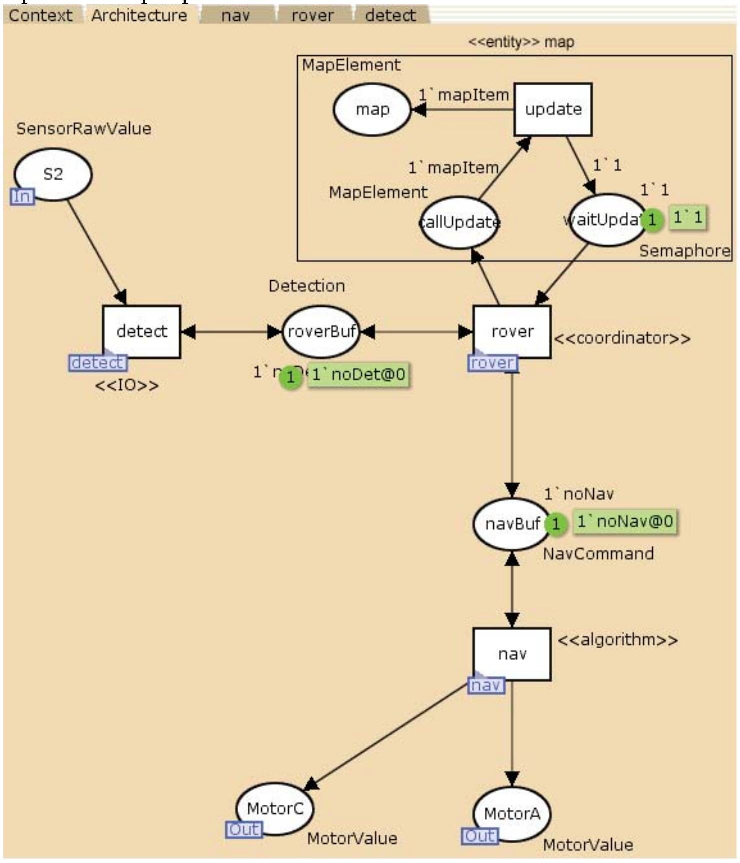


**Fig. 6.** Robot controller CPN architecture

Once an architecture-level model is established, each of the transitions representing an active object is then decomposed by applying the CPN template associated with the behavioral design pattern of that object. For the asynchronous, input-only «IO» object, “detect”, this CPN object-level model is shown in Figure 7. Here, the CPN template has been inserted and instantiated specifically for the detect object by setting the object ID to “1” as seen by the number appended to place and transition names. The specific control token, C1 has also been added as has the function for processing detections, “detection (sensorReading)”. To maintain consistency, the main transition of this template, Pin1, has also been connected to the sensor input place and to the roverBuf message buffer place.

Now, using the combined historical sizing data and information from the rover PSM, we can augment the architectural parameters within the CPN prototype to obtain further insights as to the behavioral and performance aspects that should be expected when matching the original platform independent design model with the actual platform characteristics of the target implementation.

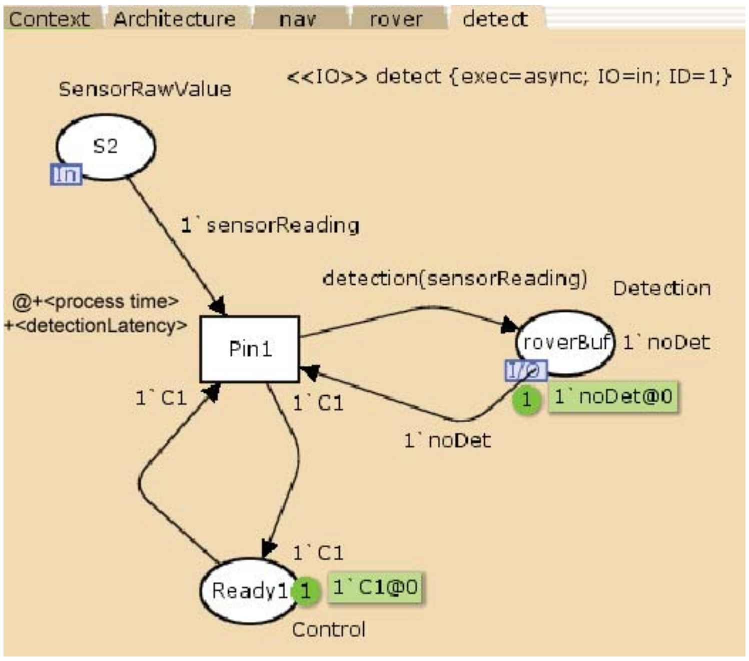


**Fig. 7.** CPN template for the "detect" object

To begin, we add a place, RAM, to our CPN prototype. This will model the available memory resources measured in bytes. The initial token value for the RAM place is calculated by subtracting the leJOS memory overhead along with the average RAM usage for the objects in our architecture from the total available RAM specified in the PSM.

**Table 1.** Calculating Memory Availability

| Source | Memory (Bytes) |
|---|---:|
| ram.sizeKB | 28,672 |
| lejos.kbMemOverhead | 17,920 |
| «IO» detect | 1,182 |
| «coordinator» rover | 2,722 |
| «algorithm» nav | 2,030 |
| «entity» map | 1,400 |
| **Available RAM:** | **3,418** |

Once the rover system begins execution, the primary consumption of memory occurs when points are added to the map object. For each point added to the map, 16 bytes are used for x and y coordinates; detection event; and timestamp. To prototype this memory consumption, the RAM place from the CPN context level model is attached to the Update transition of the map object's CPN representation on the architecture level model. This is shown in Figure 8. Using this approach, 16 bytes are subtracted from the available RAM each time the update operation is called. If the system reaches a point where less than 16 bytes are available, then the CPN model will be suspended.

Next, the <process time> parameter will be updated for each active object template. This information can be obtained from hostDemand tag in the PSM's <<psStep>> stereotype. Additionally, each «IO» object template will add the detection or output latency to the <process time>. For example, the detect object, responsible for interfacing with the light sensor, would have a basic <process time> of 10.8ms. An additional 10.3ms are then added to account for the value of lightSensor.detectionLatency from the PSM, resulting in a total time delay 21ms. Once time values have been allocated to all objects, we can move forward with analyzing the prototype of the architecture as described in the next section.
These initial estimates can eventually be replaced with higher fidelity data as it becomes available, allowing an engineer to refine the behavioral analysis as desired.

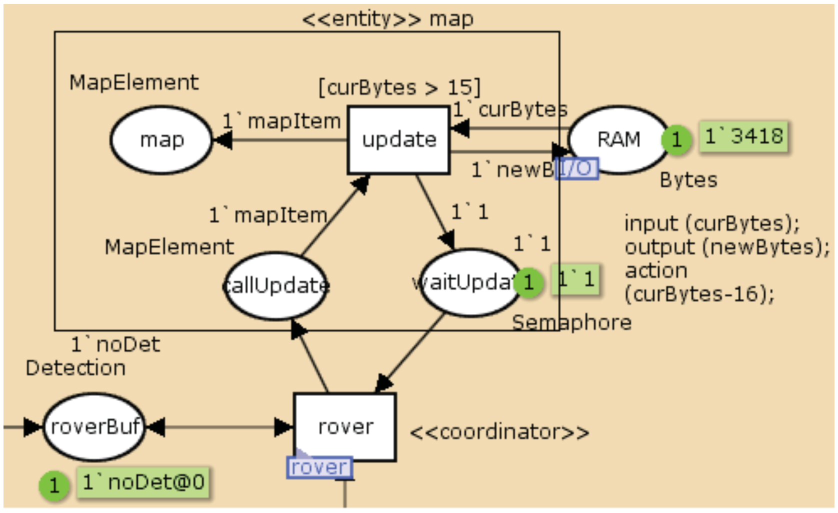


**Fig. 8.** Consumption of RAM by "map" object

## 3.4 Analyzing the Prototype

Recall from the sequence diagram of that the primary purpose of the autonomous rover system is to navigate an area, mapping objects discovered by the light sensor and taking evasive action when the light sensor detects obstacles or course boundaries. To begin analyzing this behavior with the corresponding CPN prototype, we use a test driver to provide simulated input events at random intervals. One of the first things we want to discover is how quickly the architecture responds to the detection of an obstacle or boundary. This can be analyzed from the context-level model by taking the difference in time stamps from the time an obstacle or boundary event arrives on the light sensor place to the time that a command is issued to the motors. For example, if the first obstacle was detected at time 6459 (all time is in milliseconds in this model). From the timestamps on the Motor places, we can see that from the time an input arrives to the time the system responded, there was an elapsed time of 31ms. This information could then be used, along with the speed of the rover, to determine if the reaction time is sufficient using this software architecture and this particular platform.

Other forms of analysis could include observing the memory usage over time or investigating the interaction among the concurrent objects as the simulated input rate varies. Analysis of physical architectural variations such as different light sensors or motors could also be conducted by applying different PSMs. Analysis of software architecture variations such as the use of different message communication mechanisms (e.g. FIFO or priority queuing) between the active objects could also be explored. These analyses are not shown due to space limitations in this paper.

### 3.5 Comparing Prototype with Observations

To validate our prototype, the rover design presented above was implemented in leJOS and the code was instrumented to capture timestamps. Execution of the rover when presented with boundaries or obstacles initially identified actual response times of 25-27ms from the point that the light sensor was presented with the boundary or obstacle to the point that the first motor command was output in response to the detection. This is slightly under the 31ms estimated by our analysis in the previous section. Interestingly, though, as we conducted tests with the rover over time, we observed response times increasing as battery power decreased. The above measurements of 25-27ms were observed with fully charged (9.0V) batteries. However, response times of up to 33ms were observed as the battery power was depleted to 8.2V. These results are summarized in Table 2 below. Replacing the depleted batteries with a fully charged set returned the response times to the initially observed 25-27ms. Thus, we believe that future research should include a power source with the embedded platform specific model.

**Table 2.** Response Time Results

| **Team** | **Response Time in Milliseconds** | | | |
|---|---|---|---|---|
| | **Run 1** | **Run 2** | **Run 3** | **Run 4** |
| 1 | 27 | 27 | 29 | 33 |
| 2 | 25 | 26 | 27 | 27 |
| 3 | 26 | 25 | 26 | 28 |
| 4 | 26 | 27 | 29 | 32 |
| 5 | 25 | 25 | 26 | 26 |

## 4 Conclusions and Future Research

In this paper, we have presented an approach to combine information from platform-independent and platform-specific models to construct prototypes of software architectures for embedded systems. This approach allows an engineer / analyst to examine behavioral and performance properties of a software architecture design paired with a candidate implementation architecture. The underlying CPN prototype is particularly useful in modeling concurrent object architectures in event-driven, real-time embedded systems. Applying the behavioral design patterns in the UML-based design along with corresponding CPN templates and components, the results from the analyses can be directly mapped back to the original design artifacts. Furthermore, by employing architectural parameters such as processing time, the CPN analysis model can be rapidly modified to account for different candidate architectures.

There are other tools available for constructing executable models of software designs such as the Rhapsody tool by IBM® [24]. While these tools are certainly useful, we feel there are certain advantages to our CPN approach. Modeling and prototyping of the architecture design is possible without depending on a particular UML modeling tool or design method – we only require that the basic behavioral patterns be identified. In fact, while we illustrate this approach with the UML, there is really no need to enforce the use of UML as long as the patterns can be identified for individual software abstractions. One such study applying this approach to a non-UML design is currently underway and will be published in the future. Furthermore, our CPN approach is more tolerant to varying levels of fidelity than other executable modeling tools. Using Rhapsody as an example again, each object must have detailed specifications (typically in the form of a state machine) in order for the model to be executed. With the CPN approach, even the most basic architecture designs can be simulated to show concurrent interactions, with increasing levels of fidelity as more specifications are added.

Future research in this area must continue to examine properties that should be captured and the most effective ways in which to capture them. In comparing our observed results to our analyses, the inclusion of a power model would obviously be desired in an embedded system. Additionally, future work should consider the ability to model distributed software designs configured to execute on multiple distributed embedded nodes and the communication between them. Finally, as mentioned above, work is also underway to provide a more generic approach to the executable CPN approach that allows for flexibility in the origin of the software design, whether that is captured in UML or other modeling languages.

## References

1. H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, 1 ed: Addison-Wesley, 2000.
2. R. Pettit and H. Gomaa, "Modeling behavioral design patterns of concurrent objects," presented at ICSE 2006, Shanghai, China, 2006.
3. L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada," *IEEE Computer*, vol. 23, pp. 53-62, 1990.
4. C. W. Smith, *Performance Engineering of Software Systems*: Addison Wesley, 1990.
5. H. Gomaa and D. Menascé, "Performance Engineering of Component-Based Distributed Software Systems," in *Performance Engineering 2001*, *LNCS*: Springer, 2001, pp. 40-55.

6. SEI, *A Practioner's Handbook for Real-Time Analysis - Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston: Kluwer, 1993.
7. IBM, "IBM Technical Developer," 2006.
8. Ilogix, "Ilogix Rhapsody," Ilogix, 2006.
9. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," 1996.
10. M. Baldassari, G. Bruno, and A. Castella, "PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems," *Software-Practice & Experience*, vol. 21, pp. 823-44, 1991.
11. O. Biberstein, D. Buchs, and N. Guelfi, "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism," in *COPN, Advances in Petri Nets*, *LNCS*: Springer-Verlag, 2001, pp. 73-130.
12. L. Baresi and M. Pezze, "On Formalizing UML with High-Level Petri Nets," in *COPN, Advances in Petri Nets*, *LNCS*. Berlin: Springer-Verlag, 2001, pp. 276-304.
13. K. M. Hansen, "Towards a Coloured Petri Net Profile for the Unified Modeling Language - Issues, Definition, and Implementation," Centre for Object Technology, Aarhus, Denmark, Technical Report COT/2-52-V0.1, 2001.
14. R. Pettit and H. Gomaa, "Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets," presented at 4th WICSA, Oslo, Norway, 2004.
15. B. Huber, R. Obermaisser, P. Peti, and C. E. Salloum, "Resource Specification of the DECOS Integrated Architecture," TU Wien, Vienna, Austria, Technical Report October 12, 2005 2005.
16. Lego, "Lego Mindstorms - http://mindstorms.lego.com."
17. R. Pettit, "SWE 626: Software Project Lab for Real-Time and Embedded Systems," George Mason University, 2006.
18. B. Bagnall, *Core LEGO MINDSTORMS Programming: Unleash the Power of the Java Platform*: Prentice Hall, 2002.
19. K. Proudfoot, "RCX Internals - http://graphics.stanford.edu/~kekoa/rcx/," 1999.
20. N. S. Andersen and M. N. Kjærgaard, "Advanced programming of the LEGO RCX for education," Technical University of Denmark, 2001.
21. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) Beta 2, OMG In, June 2008, http://www.omg.org/cgi-bin/doc?ptc/2008-06-08
22. UML Profile for Schedulability, Performance and Time 1.1, February 2005, OMG Inc., http://www.omg.org/technology/documents/formal/schedulability.htm
23. Unified Modeling Language (UML), Version 2.2, February 2009, OMG, http://www.uml.org.
24. IBM® Rational® Rhapsody, http://www-01.ibm.com/software/awdtools/rhapsody/