

# Languages for System Specification

Edited by  
**Christoph Grimm**

The ChDL series

Kluwer Academic Publishers

# **LANGUAGES FOR SYSTEM SPECIFICATION**

*This page intentionally left blank*

# **Languages for System Specification**

**Selected Contributions on UML, SystemC,  
SystemVerilog, Mixed-Signal Systems, and  
Property Specification from FDL'03**

Edited by

**Christoph Grimm**

*J.W. Goethe-Universität Frankfurt,  
Germany*

**KLUWER ACADEMIC PUBLISHERS**  
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 1-4020-7991-5  
Print ISBN: 1-4020-7990-7

©2004 Springer Science + Business Media, Inc.

Print ©2004 Kluwer Academic Publishers  
Dordrecht

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at: <http://www.ebooks.kluweronline.com>  
and the Springer Global Website Online at: <http://www.springeronline.com>

# Contents

Preface	ix
Part I UML-Based System Specification & Design	
<i>Introduction by Piet van der Putten</i>	
1	
UML-Based Co-Design	5
<i>Bernd Steinbach, Thomas Beierlein, Dominik Fröhlich</i>	
2	
A Unified Approach to Code Generation from Behavioral Diagrams	21
<i>Dag Björklund, Johan Lilius, Ivan Porres</i>	
3	
Platform-independent Design for Embedded Real-Time Systems	35
<i>Jinfeng Huang, Jeroen P.M. Voeten, Andre Ventevogel, Leo van Bokhoven</i>	
4	
Real-time system modeling with ACCORD/UML methodology	51
<i>Trung Hieu Phan, Sébastien Gerard and François Terrier</i>	
5	
UML-based Specification of Embedded Systems	71
<i>Mauro Prevostini, Francesco Balzarini, Atanas Nikolov Kostadinov, Srinivas Mankan, Aris Martinola, Antonio Minosi</i>	
Part II C-Based System Design	
<i>Introduction by Eugenio Villar</i>	
6	
SPACE: A Hardware/Software SystemC Modeling Platform Including an RTOS	91
<i>Jerôme Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois, El Mostapha Aboulhamid, François-Raymond Boyer</i>	
7	
LAERTE++: an Object Oriented High-level TPG for SystemC Designs	105
<i>Alessandro Fin, Franco Fummi</i>	

8

- A Case Study: SystemC-Based Design of an Industrial Exposure Control Unit 119

*Axel G. Braun, Thorsten Schubert, Martin Stark, Karsten Haug, Joachim Gerlach, Wolfgang Rosenstiel*

9

- Modeling of CSP, KPN and SR Systems with SystemC 133

*Fernando Herrera, Pablo Sánchez, Eugenio Villar*

10

- On Hardware Description in ECL 149

*Lluís Ribas, Joaquín Saiz*

### Part III Analog and Mixed-Signal Systems

*Introduction by Alain Vachoux*

11

- Rules for Analog and Mixed-Signal VHDL-AMS Modeling 169

*Joachim Haase*

12

- A VHDL-AMS library of hierarchical optoelectronic device models 183

*F. Mieyeville, M. Brière, I. O'Connor, F. Gaffiot, G. Jacquemod*

13

- Towards High-Level Synthesis from VHDL-AMS Specifications 201

*Hua Tang, Hui Zhang, Alex Doboli*

14

- Reliability simulation of electronic circuits with VHDL-AMS 217

*François Marc, Benoît Mongellaz, Yves Danto*

15

- Extending SystemC to Analog Modelling and Simulation 229

*Giorgio Biagetti, Marco Caldari, Massimo Conti, and Simone Orcioni*

### Part IV Languages for Formal Methods

*Introduction by Wolfgang Müller*

16

- Abstract State Machines 247

*Egon Börger*

17

- A New Time Extension to  $\pi$ -Calculus based on Time Consuming Transition Semantics 271

*Marco Fischer, Stefan Förster, André Windisch, Dieter Monjau, Burkhard Balser*

<i>Contents</i>	vii
18	
Modeling CHP descriptions in LTS for asynchronous circuit validation	285
<i>Menouer Boubeker, Dominique Borrione, Laurent Mounier, Antoine Siriani, Marc Renaudin</i>	
19	
Combined Formal Refinement and Model Checking for Real-Time Systems Verification	301
<i>Alexander Krupp, Wolfgang Mueller, Ian Oliver</i>	
20	
Refinement of Hybrid Systems	315
<i>Jan Romberg, Christoph Grimm</i>	
Part V Applications and new languages	
21	
Automotive Software Engineering	333
<i>Christian Salzmann, Thomas Stauner</i>	
22	
SystemVerilog	349
<i>Wolfgang Ecker, Peter Jensen, Thomas Kruse</i>	

*This page intentionally left blank*

## Preface

New applications and improved design methodologies or tools for electronic system design often require the application of new design languages. Vice versa, the availability of appropriate design languages enables new design methodologies, and allows designers to deal with new kinds of applications — often sticking on an established design language is a ‘show stopper’ for the application of new design methodologies.

An example for the impact of design languages on methodologies or tools from the past is the upcome of register transfer synthesis which was enabled by HDLs such as Conlan, and later VHDL or Verilog which allowed designers to specify the behavior of designs at register transfer level. Vice versa, these languages would not have become so popular without the availability of tools for register transfer synthesis. A recent example for the interaction of design languages and applications is the upcome of SystemC. The specification of complex HW/SW systems in HDLs, even if extended and with C interfaces, is not an appropriate solution for software developers. SystemC is one solution for this problem, and can enable new tools for architecture exploration and synthesis at higher level of abstraction.

In the future, ambient intelligence, automotive systems, etc. will combine IP cores and complex software systems with analog components such as RF or power electronics, and might have to be co-simulated with a physical environment. Established hardware description languages seem to be not appropriate for the design of such systems as long as they provide no answers to questions such as: How can we hide the growing complexity behind more abstract models? Are there approaches to hide the heterogeneity behind common models? How can we handle the verification of such systems?

This book includes selected contributions on languages for the specification, design, and verification of electronic and heterogeneous systems. These contributions give an overview of recent research topics that try to answer the above questions. The book is structured in five parts with 22

contributions and comments from Piet van der Putten, Eugenio Villar, Alain Vachoux, and Wolfgang Müller:

- The first part includes approaches to use UML and object oriented methods for the specification and design of embedded HW/SW systems.
- “C-Based system design” describes new means for and experiences in modeling HW/SW systems with C-based languages such as SystemC.
- “Analog and Mixed-Signal Systems” presents recent research activities and practical experiences in the design of heterogeneous systems.
- “Languages for Formal Methods” aims at modeling techniques that enable new techniques for verification.
- “Applications and new languages” describes requirements for design languages from automotive software engineering, and gives an overview of SystemVerilog as a new hardware description language.

All contributions were selected from the best papers presented at the “Forum on Specification and Design Languages 2003”. As organizers of this conference we would like to thank all people who supported this important event: The members of the program committee of FDL 2003, and the PC chairs Wolfgang Müller (LFM), Piet van der Putten (UML), Alain Vachoux (AMS), and Eugenio Villar (CSD) who reviewed nearly 100 submissions, and who selected the best of them. We would also like to thank Wilhelm Heupke who successfully managed the local organization together with us, and Max Slowjagin for managing correspondence with authors, and layout. Last, but not least we would like to thank Jean Mermet, Florence Pourchelle, and the ECSI staff for taking each year again the challenge to organize a new “Forum on Specification and Design Languages”.

CHRISTOPH GRIMM, KLAUS WALDSCHMIDT

I

# **UML-BASED SYSTEM SPECIFICATION & DESIGN**

*This page intentionally left blank*

Emerging highly programmable platforms enable the realization of very complex systems. The importance of the Unified Modelling Language (UML) in embedded system design is therefore growing fast. The UML workshop of FDL aims at UML-based design methods from specification to (automatic) synthesis. Such methods start from target/platform independent modelling and enable mapping on platform specific elements. From four UML sessions we selected five papers that are representative for current research activities in the field.

From the session on “Model Driven Architecture” we selected a paper by Dominik Fröhlich, Bernd Steinbach and Thomas Beierlein, called “UML-Based co-design for run-time reconfigurable architectures”. It presents a development environment based on an MDA approach, that uses UML 2.0 for specification, modelling, documentation, and visualization throughout all phases of development. The MOCCA (MOdel Compiler for reConfigurable Architectures), implements the methodology and automatically performs system validation, platform mapping, and application synthesis. Very interesting is the MOCCA action language and the mapping processes.

From the session on “Transformations & Code Generation” we selected “a unified approach to code generation from behavioral diagrams” written by Dag Björklund, Johan Lilius, Ivan Porres. The paper describes how to use ‘Rialto’ intermediate language to capture the semantics of UML behavioral diagrams, such as statechart, activity and interaction diagrams which is an interesting work because Rialto has a formal semantics given as structural operational rules and supports semantic variations. It can be used to uniformly describe the behavior of a combination of several diagrams and as a bridge from UML models to animation and production code.

From session “Real-Time and System-level modelling” we selected two papers. The first paper is by Jinfeng Huang, Jeroen P.M. Voeten, Andre Ventevogel and Leo van Bokhoven, called “Platform-independent design for embedded real-time systems”. It is an important contribution because it describes essential characteristics that current tools miss. It proposes an approach based on both platform-independent design and correctness-preserving transformation. It is research in the context of the real-time modelling language ‘POOSL’, and the tool ‘rotalumis’ that automatically transforms POOSL models into executable code for target platforms, preserving the correctness properties verified in the model.

The second paper from session “Real-Time and System-level modelling” is from Trung Hieu Phan, Sébastien Gerard and François Terrier,

called “Real-time system modeling with accord/uml methodology”. The method is illustrated on an automotive case that has multitasking and real-time constraints. The ACCORD/UML method proposes UML extensions for modeling real-time systems and follows the MDA paradigm. Behaviour can be expressed early in the model itself and then implemented automatically.

The fifth paper from the session “UML for Hardware/Software co-design” by Mauro Prevostini, et al., called “UML-based specifications of an embedded system oriented to hw/sw partitioning” proposes an approach to define hardware/software partitioning of an embedded system starting from its UML system specifications. The approach is illustrated with a case study, a Wireless Meter Reader (WMR) dedicated to the measurement of energy consumption.

Piet van der Putten

*Technische Universiteit Eindhoven  
Department of Electrical Engineering  
p.h.a.v.d.putten@tue.nl*

# Chapter 1

## UML-BASED CO-DESIGN FOR RUN-TIME RECONFIGURABLE ARCHITECTURES

Bernd Steinbach<sup>1</sup>, Thomas Beierlein<sup>2</sup>, Dominik Fröhlich<sup>1,2</sup>

<sup>1</sup>*TU Bergakademie Freiberg  
Institute of Computer Science*

<sup>2</sup>*Hochschule Mittweida (FH) - University of Applied Sciences  
Department Information Technology & Electrotechnique*

**Abstract** In this article we present an object-oriented approach and a development environment for the system-level design of run-time reconfigurable computer systems. We use the Unified Modelling Language (UML) for the specification, modelling, documentation, and visualization throughout all phases of development, from specification to synthesis. The proposed approach is based on hardware-software co-design and Model Driven Architecture (MDA). This way we allow for thorough and complete system representations, platform-independence, comprehensible and seamless transition from specification to implementation, and the description of common development artifacts and activities. In this article we will focus on aspects and problems which are related to object-orientation, UML, and MDA.

**Keywords:** Unified Modeling Language, Model Driven Architecture, Platform Mapping, Synthesis

### 1. Introduction

The explosion of computational highly demanding applications drives the development of more powerful microprocessor architectures. However, the performance needs of these applications is growing much faster than the rate of improvement in processor speed. Further the ratio of speedup to the number of deployed transistors is steadily decreasing.

Thus new computer system designs are evolving. A promising new class of architectures are *run-time reconfigurable computer systems* (RTR).

Run-time reconfigurable computer systems generally combine a classical microprocessor (uP) with programmable logic resources, like field programmable gate arrays (FPGA). The uP executes the global control flow and those parts of the application that would require too much effort if implemented in hardware. CPU-intensive parts of the application or critical I/O operations are performed by the FPGAs. The functionality implemented by the logic resources is dynamically adapted to the executed algorithm. Recently speedups of two orders of magnitude compared to classical computer systems have been reported for reconfigurable computer systems, for instance [2]. What makes these systems appealing is their unique combination of cost per speedup and flexibility.

However, years after its inception, reconfigurable computing still faces a wide gap between the capabilities offered by the implementation technology and the availability of mature methodologies and tools for the development of their targeted applications. For reconfigurable computing to move out of the laboratories into general purpose computing, time and cost for the application design, implementation and deployment must become comparable to software development. The suitability of the employed development methodology and the capability to exploit the flexibility of the hardware will have more impact on RTR systems, than the optimality of the implementation, in terms of the reached cost or speedup.

We developed a methodology and a supporting tool for the object-oriented system-level specification, design, and implementation of applications for RTR architectures. In our approach the system being developed and the target platform are specified with models. The models are described with the Unified Modelling Language (UML), version 2.0, and a dedicated action language [7]. Our methodology defines the development steps to transform the system specification into a ready-to-run application. A UML model compiler, MOCCA (MOdel Compiler for reConfigurable Architectures), implements our methodology in that it automatically performs system validation, platform mapping, and application synthesis. The synthesized application exploits the potential performance gains offered by the reconfigurable hardware. The MOCCA environment was not designed for a special class of applications, however we envisage the acceleration of computationally intensive algorithms from scientific computing, automation, and communication, like boolean problems, neural networks, track generation, and data compression.

The research on development methodologies for reconfigurable computer systems is still quite new. Chata and Vemuri presented the SPARCS design environment and a design flow [1]. However, they do not address system-level specification, but high-level behavior specifications. Eisenring and Platzner are developing an implementation framework for RTR systems, which consists of a development methodology and a design representation [4]. The system specification is done at the task-level. The framework allows for partitioning, estimation, and synthesis. Vasilko et al. developed a modelling and implementation technique which builds upon the OCAPI-XL design environment [9]. The system is specified with processes communicating through semaphores and mailboxes. The processes that can be refined to lower level descriptions which are then realized either in hardware or software. In contrast to these approaches we specify systems with objects communicating through messages.

In the rest of this article we will present our development approach. The approach is motivated and outlined in section 1.2. Section 1.3 introduces system modelling and the action language. In section 1.4 we briefly discuss platform mapping, and in 1.5 synthesis is explained. In the course of discussion we will briefly introduce how each activity is supported by the MOCCA-environment. Finally, in section 1.6 the article is concluded.

## 2. UML-Based Co-Design Approach

### 2.1 Motivation

It is common practice today to use software programming languages, like Java, C, or C++, or variants of programming languages to capture system-level specifications, e.g. SystemC and SpecC [13], [10]. Executable co-specifications are commonly used as basis for system validation and synthesis. However, these languages have a strong focus on implementation. This renders them actually insufficient for the purpose of system-level specification, because

- the different aspects of a system, e.g. requirements, design, implementation, and deployment, are not sufficiently separated. For instance, the system function is captured in one of many possible implementations.
- some important system aspects can not be represented adequately. For instance, often the specification of non-functional requirements, design constraints, and deployment information is captured in code, comments, and/or compiler directives.

- the specification is inherently platform and technology dependent. Examples are dependencies on specific hardware, operating systems, programming language semantics, physical data organization, and prerequisite libraries.

Due to these problems, there is a growing interest in using UML as co-design language. The use of UML has several advantages in comparison to programming language based approaches: First, in UML based approaches systems are specified by means of models rather than code. This may improve design quality and tractability of complex systems. Second, the expressive power and generality of UML allows for the specification of the important system aspects, like requirements, design, and deployment, in a consistent and comprehensive manner to the user. Third, the extensibility of UML makes the language adoptable to different application domains.

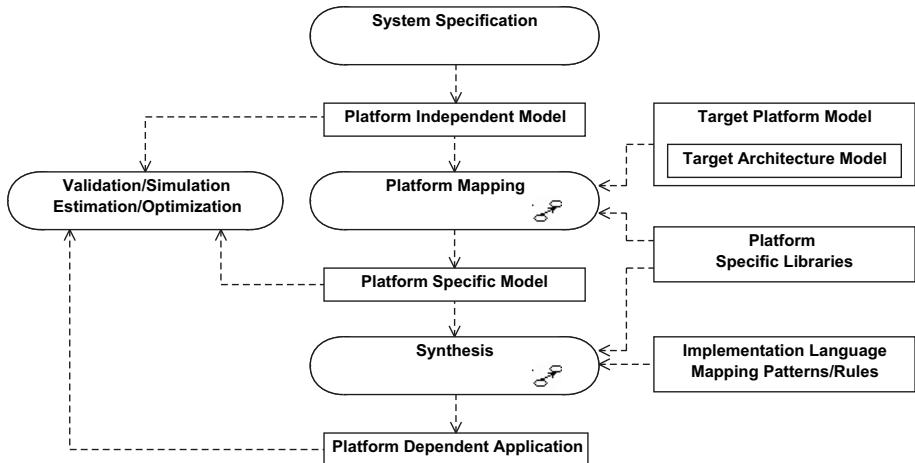
As with UML 2.0 the formerly crude capabilities for the detailed modelling of behavior improved significantly. The UML action semantics enable the complete and precise behavior specification. This is a premise for model execution and interchange. Action languages provide a syntax for this semantic which is suitable to the targeted application domains. Currently, there is already a number of action languages tailored to software-only applications, but we expect that there will be special co-design languages in the near future.

## **2.2 Activities and Artifacts**

Figure 1.1 shows the basic activities and artifacts of our development approach. The illustrated approach incorporates the specify-explore-refine paradigm of hardware-software co-design into the concept of Model Driven Architecture [5], [8], [6]. It is a particular goal of MDA to separate the application from the platform, comprising of hardware, operating systems, compilers etc., used for its implementation. This not only supports the retargeting of the application but can also simplify the mapping of a given application to a particular platform.

In our approach both, the system and the target platform are defined by means of independent models, described with UML and an action language. The system is specified by means of a platform independent model (PIM). The target platform model (TPM) defines the services provided by the implementation target for system execution.

During platform mapping the platform independent model is transformed into a platform specific model (PSM). During synthesis the system is synthesized by transforming the PSM into a platform dependent application (PDA) which can be executed on the target platform. The



*Figure 1.1. UML-Based Co-Design - Activities and Artifacts*

application and its models may be validated and simulated. Additionally estimation, e.g. of implementation and execution characteristics, and optimizations may be performed.

The presented methodology is neither specific to run-time reconfigurable architectures nor to UML. However, in the MOCCA-environment it was implemented specifically for RTR-architectures and UML.

The following sections provide a more thorough discussion of the important development activities and artifacts. For more information and examples we refer the reader to [12] and [11].

### 3. System Specification

#### 3.1 Platform Independent Model

The system is specified in a platform independent model of its function, structure, and behavior. The PIM comprises the following models:

- an use-case model, which represents the system function by means of the services it provides to the system context.
- a design model, which represents an executable and platform independent realization of the use cases as specified by the use-case model. The design model may define implementation constraints and execution information.

System modelling is finished when the design model successfully executes all test cases defined for the realized use cases. System function is specified with use cases and collaborations. To specify the structure subsystems, classes, interfaces, and their relationships are used. Behavior specification is done with state machines, methods, and actions. The model of computation is objects communicating through synchronous method calls and asynchronous signals. Method calls, signal sends, and the events they evoke are polymorphic, following similar rules as specified in [6]. For the detailed description of behavior we use the MOCCA action language (MAL). A thorough discussion of MAL is beyond the scope of this article.

### **3.2 MOCCA Action Language**

The UML action semantics are a premise for building executable and interchangeable UML-models. An action language should allow for full model access, abstract from physical data organization, and not overemphasize sequential execution. The adherence to these criteria is considered a key property of system level specification languages. For an action language to be suitable to hardware/software co-design, it must allow for

- model execution
- early verification and simulation
- hardware-software design space exploration
- automatic hardware and software synthesis

at the system-level of abstraction. For this, the language should facilitate analysis and optimization of control and data flow, and estimation of non-functional system characteristics, like performance, implementation effort, and power consumption. The need for accurate estimates suggests to remain as close as possible to the control flow and data organization of the synthesized application.

Existing action languages, like ASL, AL, and SMALL [6] are designed for software-only applications in business, telecommunication, and network systems. They have a high level of abstraction, in that they do not make any assumptions about data organization. Although this provides some potential for performance optimization, it complicates estimation.

Thus we designed the MOCCA action language. This language is compliant to the UML action semantic specification. In comparison to the named action languages it has a medium level of abstraction in that it requires the developer to make data organization and data access explicit.

However, this allows us to employ standard analysis, optimization, and estimation techniques. Nevertheless, we consider the support of higher level action languages a major research issue for the future, because they make much of the data concurrency explicit. MAL allows for the specification of sequential logic, arithmetical/logical relational operations, instance and link manipulation, class access, and time calculation.

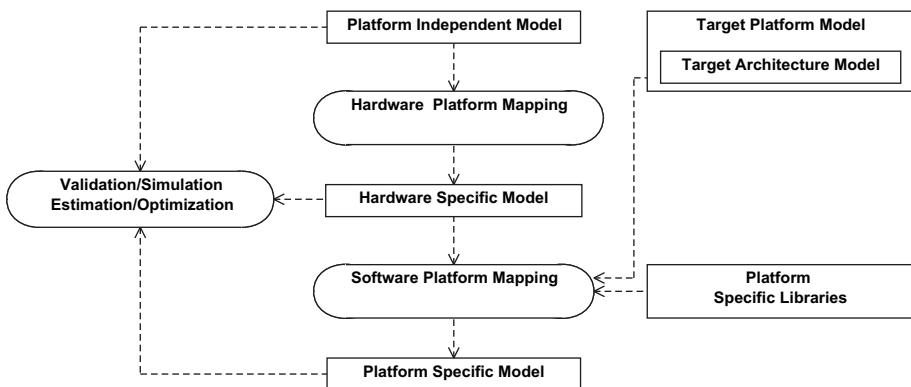
To make the language easy to use and to reduce the learning effort, the syntax and semantics of the language orients towards the statement and expressions of the Java programming language. The language does not overlap with concepts that already exist in the UML. However, it supports important UML concepts not existing in Java, like associations, active classes, state, and time.

## 4. Platform Mapping

### 4.1 Activities and Artifacts

Given a PIM or a partial version of the PSM we proceed by mapping it to the target platform. Throughout this platform mapping step we transform the system model such that it can be implemented on the target platform and the resulting application optimally exploits the platform resources.

As illustrated in Figure 1.2, platform mapping is performed such that hardware platform mapping is performed before software platform mapping.



*Figure 1.2. Platform Mapping - Activities and Artifacts*

First, throughout hardware platform mapping, basic behavior, as specified with UML actions, is partitioned among the hardware components.

The result of this step is a hardware specific model of the system (HSM). During software platform mapping this model is transformed to a PSM, by mapping the behaviors which require additional run-time software support to the respective resource services.

## 4.2 Target Platform Model

The target platform model (TPM) specifies all information necessary for complete and automated platform mapping and synthesis:

- the target architecture model (TAM) in which the resources and resource services provided by the platform for application implementation are modelled. Resources are hardware devices and execution environments, like reconfigurable devices, microprocessors, communication paths, storages, and run-time environments. The resource services of each resource comprise an abstract operation set which defines the resource's functionality. Each resource service is characterized by its implementations and their QoS-characteristic (quality-of-service), like latency, area, and power consumption. Services provided by resource types, like storages or communication paths, are modelled similarly. Additionally, for each abstract operation implementation, implementation language specific rules and patterns, and necessary libraries are given.
- the model compiler components to be used for hardware resource specific platform mapping, estimation, generation, as well as the interpreter of the generated output, like synthesis or compilation tools, are modelled in the TPM. By means of these components the model compiler dynamically adapts its back-end to the hardware architecture.

UML 2.0 enables us to describe the TPM entirely with UML and MAL.

## 4.3 Hardware Platform Mapping

**Activities and Artifacts.** Hardware platform mapping is the transformation of the PIM (or a partial PSM) of the system into a HSM. During hardware platform mapping we identify a partitioning of the system function among the components of the target hardware architecture. It is performed by iteratively changing the model structure and behavior such that it can be implemented on the given target platform. Model transformations are described with mapping operators. The design space, as defined by the target platform, is explored for feasible implementations by the iterative application of the mapping operators to the model.

The structure and representation of the HSM is discussed in section 1.4.3.0. Hardware platform mapping algorithms and the mapping operators are briefly discussed in section 1.4.3.0. For the purpose of this article we will focus on the language related aspects.

**Hardware Specific Model.** The hardware specific model defines the partitioning of the system function among the resources of the target platform by means of establishing relations between the elements of the PIM and the according elements of the TPM being used to realize the PIM elements. The HSM depicted in Figure 1.2 comprises of the following models (due to space limitations not shown in the figure):

- an implementation model, which represents the mapping of design model subsystems and classes to implementation subsystems and classes and their mapping to components. For each implemented element of the PIM this model specifies which resource services are allocated for its implementation.
- a deployment model, which represents the mapping of the components of the implementation model to the components of the target architecture model.

The implementation and deployment model are described in the UML and MAL. That is, the HSM can be executed, verified, and visualized using the same algorithms and tools as for the PIM. Moreover, this allows the developer for a (semi-) manual partitioning by providing complete or partial versions of these models. Partitions computed by the model compiler can be changed or visualized using ordinary UML modelling tools.

**Mapping Algorithms.** We based partitioning on objects, mainly for two reasons. First, objects are a fundamental source of concurrency. Each object can be seen as an abstract machine providing services that may be requested by other objects by means of messages. All objects of a system execute concurrently; synchronization is reached through the messages. This object-based model of computation maps well to RTR-architectures comprising of multiple, possibly reconfigurable, processing elements. Second, objects encapsulate data and behavior to process the data. During partitioning of objects, both function and data are partitioned, which can reduce communication effort.

To transform the PIM into a PSM we defined a set of mapping operators. We distinguish three classes of transformations: allocations, behavior transformations, and structure transformations. Allocation operators act on all realized model elements. They map model elements

to the target architecture by assigning sufficient sets of resource services. Behavior transformations operate on UML behavior definitions, like state machines and activities, and the model elements which implement them. Structure transformation operators operate on model elements from which the model structure is built. Table 1.1 describes the mapping operators.

*Table 1.1.* Mapping Operators

Operator	Description
<b>Allocation Operators</b>	
ALLOCATE/DEALLOCATE-(Resource Service Set, Element)	Allocates or deallocates a set of resource services from the element. In case of ALLOCATE the element is realized with the resource services in the set.
<b>Behavior Transformation Operators</b>	
DECOMPOSE(Behavior)	Splits the given behavior into a set of behaviors that collaboratively implement the same functionality as the original behavior. This operator must be used in conjunction with IMPLEMENT(Behavior, Operation/Class), and JOIN(Operation, Class) in case of the former, in order for the resulting model to be valid.
IMPLEMENT(Behavior, Operation)	Associates the behavior to the operation. The behavior defines the implementation of that operation. The operation provides the interface to the behavior.
IMPLEMENT(Behavior, Class)	Associates the behavior to the class. The behavior defines the states the instances of the class may be in during their life-time.
<b>Structure Transformation Operators</b>	
JOIN/SPLIT(Feature, Class)	Join (split) a feature, like attribute or operation, to (from) a class. In case of JOIN the class will encapsulate the feature.
JOIN/SPLIT(Class, Component)	Join (split) a class to (from) a component. In case of JOIN the component will realize the class.
JOIN/SPLIT(Component, Device)	Join (split) a component to (from) a device. In case of JOIN the component is deployed on the device.

The type, operands, sequence, and the degree of automation of the operator application is subject to the concrete mapping algorithm. Given this framework one can think of many different mapping algorithms. The feasibility of an algorithm depends on its specific trade-off between the quality of the computed solution and the effort to compute this solution, which not only depends on the mapping algorithm itself but also on its input - a PIM or a partial solution of a PSM, and the TPM.

In general mapping algorithms comprise of three major steps: computation and pre-estimation of candidate mappings, selection of mapping sets, and application of mapping sets. Candidate mappings are computed for each realized model element. They are described in terms of the presented mappings operators. The number and type of candidate mappings being computed for a specific model element is a function of the element's expected utilization. For instance, for model elements the number of pre-computed allocation mapping candidates may be a function of its execution probability and frequency [3]. Thereby we account for the different implementation options of each element. Moreover, this allows us to adjust the trade-off between compilation time and the quality of the computed model mapping. The local QoS of each candidate mapping is preestimated using the QoS information modelled in the TPM.

The actual mapping is performed by selecting and applying a set of candidate mappings to the PIM or a partial version of the PSM. We start by computing a complete PSM, which is then progressively refined such that the quality of the overall mapping is optimized. For this we iteratively select sufficient sub-sets of candidate mappings and estimate the effect of their application to the model. Then we decide if the mappings shall be applied, and if so, we apply them. The compiler validates that at each step the selected set of mappings is consistent and complete, in that none of the mappings affects another mapping and the resulting PSM describes a complete mapping. The strategy for the selection and application of mapping sets is subject to the concrete mapping algorithm. For instance a simulated annealing algorithm will typically select candidate mappings for only a small number of cohesive model elements and apply those mappings which improve the current solution. A branch-and-bound algorithm may apply the mappings in a deterministic order and apply only the best.

We implemented the described framework in the MOCCA-compiler. Currently, MOCCA provides six mappers which employ different strategies for mapping selection and application, like randomized search or branch-and-bound. The mappers are parameterizable and may be combined with each other such that a PSM computed by one mapper is the initial solution for the next mapper.

## 4.4 Software Platform Mapping

Given a hardware specific system model, that is a model in which parts of the behavior are mapped to the suitable hardware components, we proceed by completing the platform specific model of our system.

The basic behavior of the system is specified with UML actions. In addition each non-trivial UML model contains behavior, which is and can not be described with actions alone. The realization of this behavior mostly requires services provided by the underlying run-time environment and synthesis libraries. It is important to note, that in the PIM we are not to specify such realizations, because this would violate platform independence. It is the task of the software platform mapping step to bind these behavior to the resources required for their actual realizations. Examples of such mappings are

- active classes are mapped to threads/processes of the underlying operating system.
- operation guards are mapped to mutexes/semaphores of the underlying operating system.
- state machines may require a storage and access mechanism for events. Further run-time support may be necessary depending on their actual realization (table driven, direct encoding, ... )
- associations with multiplicities greater than one require a storage and access mechanism for the references to associated objects.

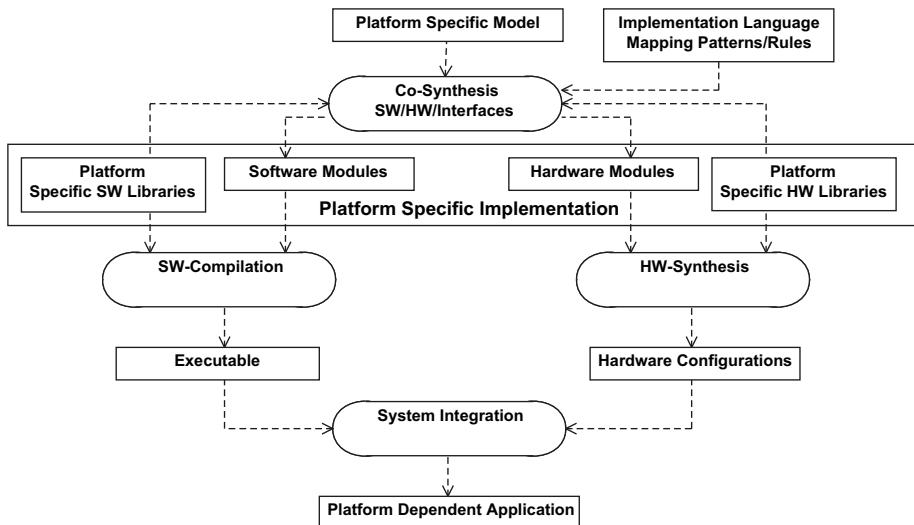
The resources which comprise the software platform are described in the target platform description. For each respective model element the model compiler selects the required resource and binds it to the model element. The binding information is transparently included in the HSM by means of UML extensions mechanisms, so, if required, it may be changed by the developer.

## **5. Synthesis**

Given a platform specific model of our system, we proceed by implementing it into an ready-to-run application. Figure 1.3 shows the activities and artifacts of this step.

Central activity is the synthesis of the platform specific implementation, comprising of hardware and software modules and communication interfaces according to the platform mapping. Synthesis is performed on the basis of the implementation and deployment models. For the elements realizing UML components we synthesize an according implementation. The implementation language is specific to the processing element on which the component will be deployed.

Synthesis depends on the used implementation language and language specific transformation patterns and rules. In our approach these are incorporated in the language specific back-ends of the model compiler. We



*Figure 1.3. Synthesis - Activities and Artifacts*

currently support C++ and Java for implementation of software components and VHDL and Java for the Xilinx Forge compiler for hardware components [14].

For detailed synthesis/compilation to configuration bit-streams/executables we use commercial tools. Synthesis and compilation are parameterized with the design constraints in the model. After synthesis and compilation we integrate the bit-streams and executables into the application which can directly be executed on the target platform.

## 6. Conclusions and Future Work

In this article we presented an approach and a supporting environment for the object-oriented system-level development of applications for runtime reconfigurable computer systems. To overcome some of the drawbacks of approaches which use conventional programming languages for system specification, we use the universal syntactic and semantic framework provided by the Unified Modelling Language. To enable model execution, early verification and simulation, design space exploration, and automated and complete synthesis we extended the UML by an appropriate action language.

We have shown that this approach enables a seamless and comprehensible transition of the system specification to a ready-to-run application, without the need to change the applied paradigms, methods, or language

in between. Also the presented approach defines a comprehensible development process, which incorporates the hardware-software co-design into the general approach of Model Driven Architecture. Although UML and MDA were not explicitly designed for the development of hardware-software systems, we beneficially utilize them.

The development process and the MOCCA model compiler as presented in this article are functional [11]. The flexible architecture of the model compiler makes it retargetable to different input specification languages and representations, target platforms, and development environments. Future will include the development of platform mapping algorithms and accompanying heuristics specialized to object-oriented systems.

## References

- [1] Chata, K. S. and Vemuri, R. (1999). Hardware-Software Codesign for Dynamically Reconfigurable Systems. In *Proceedings of the 9. International Conference on Field Programmable Logic and Applications (FPL'99)*.
- [2] Dandalis, A., Prasanna, V. K., and Thiruvengadam, B. (2001). Run-Time Performance Optimization of an FPGA-Based Deduction Engine for SAT-Solvers. In *Proceedings of the 11. International Conference on Field Programmable Logic and Applications (FPL'01)*, pages 315–325.
- [3] Drost, I. (2003). Estimation of Execution Probabilities and Frequencies of OO-Models. Diploma Thesis. University of Applied Sciences Mittweida, Germany.
- [4] Eisenring, M. and Platzner, M. (2000). An implementation framework for run-time reconfigurable systems. In *The 2nd International Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE'00)*. Monte Carlo Resort. Las Vegas, USA.
- [5] Gajski, D. D. and Vahid, F. (1995). Specification and Design of Embedded Hardware-Software Systems. *IEEE Design and Test of Computers*, pages 53–66.
- [6] Mellor, S. J. and Balcer, M. J. (2002). *Executable UML - A Foundation for Model Driven Architecture*. Addison Wesley Longman Publishers.
- [7] Object Management Group (2003). OMG Unified Modelling Language Specification (Super Structure). <http://www.omg.org>. Version 2.0.

- [8] Object Management Group - Architecture Board ORMSC (2001). Model driven architecture - a technical perspective (MDA). <http://www.omg.org>. Draft.
- [9] Rissa, T., Vasilko, M., and Niittylahti, J. (2002). System-level modelling and implementation technique for run-time reconfigurable systems. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*.
- [10] SpecC Technology Open Consortium (2002). SpecC Language Reference Manual, Version 2.0. Specification. <http://www.specc.org>.
- [11] Steinbach, B., Beierlein, T., and Fröhlich, D. (2003a). The MOCCA-compiler for run-time reconfigurable architectures. Mocca Project Web-Pages.  
<http://www.htwm.de/lec/mocca> (under construction).
- [12] Steinbach, B., Beierlein, T., and Fröhlich, D. (2003b). UML-Based Co-Design of Reconfigurable Architectures. In *Proceedings of the Forum on Specification and Design Languages (FDL'03)*, Frankfurt a.M., Germany.
- [13] SystemC Language Working Group (2002). Functional specification for systemc 2.0. Specification. <http://www.systemc.org>.
- [14] Xilinx Inc. (2003). Forge - compiler for high-level language design.  
<http://www.xilinx.com/ise/advanced/forge.htm>.

*This page intentionally left blank*

## Chapter 2

# A UNIFIED APPROACH TO CODE GENERATION FROM BEHAVIORAL DIAGRAMS

Dag Björklund, Johan Lilius, Ivan Porres

*Turku Centre for Computer Science (TUCS)*

*Department of Computer Science, Åbo Akademi University*

*Lemminkäisenkatu 14 A, FIN-20520*

*Turku, Finland*

**Abstract** In this article we show how to use the Rialto intermediate language, to capture the semantics of UML behavioral diagrams. The Rialto language has a formal semantics given as structural operational rules and it supports semantic variations. It can be used to uniformly describe the behavior of a combination of several diagrams and as a bridge from UML models to animation and production code.

### 1. Introduction

The Unified Modeling Language (UML) [1] can be used to model the architecture and behavior of any kind of software project. This is due to the fact that UML provides many different diagrams or views of a system: class, component and deployment diagrams focus on different aspects of the structure of a system while the behavioral diagrams such as use case, statechart, activity and interaction diagrams focus on its dynamics.

All the behavioral diagrams except use case diagrams are closely related. We can convert a collaboration diagram into a sequence diagram and vice versa. Statecharts are used as the semantic foundation of the activity diagrams and it is possible to represent an execution (a trace) of a statechart or an activity diagram as a sequence or collaboration diagram. However, at the formal level, we consider that the UML standard lacks a consistent and unified description of the dynamics of a system specified using the previous diagrams.

Many authors are working towards a formal semantics for UML [2]. Formal semantics give an unambiguous interpretation of UML using a mathematical formalism; some of the recent works in this direction are the semantics for statecharts e.g. [3] and [4] and activity diagrams [5]. Although these articles are sound and inspiring, they do not study the semantics of multiple diagrams combined.

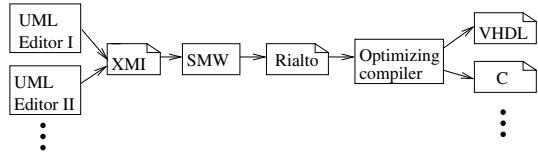
In this paper, we show how we can formalize the behavior of UML diagrams, including the combined behavior of several diagrams, using Rialto. Rialto is an intermediate language that can be used to describe multiple models of computation. A model of computation is a domain specific, often intuitive, understanding of how the computations in that domain are done: it encompasses the designer's notion of physical processes, or the "laws of physics" that govern component interactions. UML statecharts (a chapter in [1]) is an example of an asynchronous model of computation while languages like ESTEREL [6] or Harel's Statecharts [7] have semantics based on the synchronicity hypothesis (a variant of the synchronous computational model).

We have had as our starting point the observation that many language constructs are common to different languages, but their semantics differ due to the computational model. E.g. parallelism in UML has a very different semantics from parallelism in ESTEREL. In [8], Lee touches the subject of decoupling an abstract syntax from the model of computation, i.e. he suggests that a language, or a set of languages, with a given abstract syntax, can be used to model very different things depending on the semantics and the model of computation connected to the syntax. In the context of UML, this phenomenon can be observed e.g. with statecharts and activity diagrams. An activity diagram can be interpreted as a statechart where all computation is done in state activities and the transitions are triggered by completion events. Therefore, we can say that activity diagrams have data flow as their underlying model of computation. In Rialto, we define different scheduling policies for different computational models. The policies define atomic execution steps and hence we can for instance model synchronicity in a way similar to that of SystemC for instance. It is this interaction between language constructs and models of computation that is the focus of our research and our motivation for introducing Rialto as a language for describing the behavior of UML.

We are also applying this new insight on the combined behavior of UML models to construct tools to animate the models and generate optimized code. Model animation enables a designer to validate a model before the design and implementation has been completed. Rialto can be used as an execution engine for UML models, and is hence also related

to the work on executable UML [9]. Rialto can also be used to generate code for different target languages, including C and VHDL.

The translation from UML models to Rialto is automatically performed using the SMW toolkit [10] as shown in the figure.



The SMW toolkit can be used to transform and extract information from UML models. SMW can read UML models created by any XMI-compliant UML editor. The generated Rialto code is compiled into target language code by the Rialto compiler. The whole process can be made totally transparent and we have created prototype model compilers like `uml2cpp` and `uml2vhdl` that read in XMI files and output C++ and VHDL code.

This article is divided into three parts: We start by introducing Rialto, our intermediate language, then we explain how we represent UML statecharts, activity diagrams and collaboration diagrams in this language. Finally, we describe our strategy for code generation from Rialto to target language code.

## 2. The Rialto Intermediate Language

Rialto is a textual language for multiple models of computation with formal semantics. It can be used as a stable platform for interfacing UML behavioral models with tools for code synthesis, animation, verification etc. We do not expect the average UML practitioner to use Rialto. Instead, we intend it to be both a tool to study the generation of efficient code and a tool for UML scholars to discuss different aspects of UML behavioral semantics. The Rialto language is more abstract than usual programming languages since it supports concepts like traps (high-level transitions), suspension and resuming of threads, and event queues. The language has been specifically designed to describe the behavior of modeling languages and it can be used to combine multiple heterogeneous models of computation [12].

In this section we shortly present some syntactic elements of Rialto, give a semantics for the statements in terms of structural operational rules and show how we can use different scheduling policies to give different behavior to different parts of a system.

### 2.1 Syntax

Every statement in a Rialto program has a unique label, which can e.g. correspond to the name of some UML element, or it can just be a unique

name given by the precompiler of the language. The key abstraction in the language is the **state** block. It can represent different entities in different domains, it also represents a scope that can be assigned a scheduling policy, as described below. A **state** block is declared as follows:

```
state
    # declarations, traps and scheduling policy
begin
    # substates, sequential code
end
```

If we plan to use the same state block more than once, we can define a **stateType** to avoid code duplication. A **stateType** is declared as follows:

```
typename: stateType( ([in|out] port)* )
    # state definition
end
```

Where the “state definition” is a normal **state** block and the ports are channels used by the **stateType** to communicate with other entities. A **stateType** is instantiated as: **label:** **typename( q1 (, q2 )\*** ).

We have included in the language some concepts that are common to different models of computation, but can have different semantics depending on different computational models. We will now give a short description of some of the concepts and show how they are represented in Rialto.

**State** A state in a UML or Harel’s statechart can be represented directly in Rialto by a **state** block. As in UML statecharts, Rialto **states** can contain substates, which can also be orthogonal. **Interrupts and transitions** An interrupt is an event of high priority. In our language, a **trap** statement is used to monitor interrupts. Interrupts correspond to **trap** in ESTEREL and transitions going upwards in the state hierarchy in both UML and Harel’s Statecharts. The **goto** statement is used to do transitions from a state to another. A **trap** that monitors an event **e** on a queue **q** and does a transition to a state **A** is declared as follows:  
**trap q.e do goto(A).**

**Concurrency** Concurrency means that several statements are executed in parallel. In our language, concurrency is defined using the **par** statement, where the arguments are the labels of, usually **state** statements, which should be run in parallel. The scheduling policies manage concurrent threads, which reflects the computational model.

**Thread Synchronization** Concurrent threads in a system often need to be synchronized, for instance two parallel activities in an ac-

$[par]$	$\frac{l \in \alpha \quad \mathcal{L}(l) = \text{par}(l_1, l_2, \dots, l_n)}{\langle \alpha, \mathcal{E} \rangle \xrightarrow{l:\text{par}(l_1, l_2, \dots, l_n)} \langle \alpha - \{l\} \cup_{i=1}^n \{l_i\}, \mathcal{E} \rangle}$
$[goto]$	$\frac{l \in \alpha \quad \mathcal{L}(l) = \text{goto}(l_1, \dots, l_n)}{\langle \alpha \rangle \xrightarrow{g\text{o}\text{t}\text{o}(l_1, \dots, l_n)} \langle \alpha - \text{subtree}(l, l_1, \dots, l_n) \cup \{l_1, \dots, l_n\} \cup \text{entry-stats} \rangle}$
$[trap_1]$	$\frac{l \in \alpha \quad b \quad \mathcal{L}(l) = \text{trap } b \text{ do } S_1 \quad \langle \alpha \cup \{\mathcal{L}^{-1}[S_1]\}, \mathcal{E} \rangle \xrightarrow{S_1} \langle \alpha', \mathcal{E}' \rangle}{\langle \alpha, \mathcal{E} \rangle \xrightarrow{l:\text{trap}} \langle \alpha', \mathcal{E}' \rangle}$

Table 2.1. A subset of the operational rules of the language

tivity graph are synchronized by a join pseudo-state. In Rialto we can specify a synchronization boundary by encapsulating threads within a **state** block, which activates its successor when each of the contained threads has finished.

**Communication policy** A communication policy states how different modules of the system communicate with each other. The communication mechanism we use with UML statecharts is a simple fifo queue. In state diagrams, an event is an occurrence that may trigger a state transition. In UML statecharts, there is an implicit global event queue.

## 2.2 Operational Semantics

To define the operational semantics of the language we need to formalize the structure of a Rialto program. The labels of the statements act as values for program counters. The set of labels  $\Sigma$  is organized in a tree structure by the *child* relation  $\downarrow$  (The tuple  $\langle \Sigma, \downarrow \rangle$  forms a label-tree). This reflects the scoping of the program. We can take the closure  $\downarrow^*$  of  $\downarrow$ : let  $l_1$  and  $l_2$  be labels; then  $l_1 \downarrow^* l_2$  means  $l_1$  is a *descendant* of  $l_2$  or in other words:  $l_1$  is inside the scope of  $l_2$ . The *successor* relation  $\succ \subset \Sigma \times \Sigma$  orders the labels sequentially, If  $l_1, l_2 \in \Sigma$  and  $l_1 \succ l_2$  then  $l_2$  succeeds  $l_1$  in the program.  $\mathcal{L} : \Sigma \rightarrow \text{statement}$  is a total function that maps each label to its statement.

We can now identify a program by the tuple  $\langle \Sigma, \downarrow, \succ, \mathcal{L} \rangle$ . The state configuration of a program is represented by a tuple  $\sigma = \langle \alpha, \gamma, \mathcal{E} \rangle$ , where  $\alpha \subseteq \Sigma$  is the set of active labels,  $\gamma$  is the set of suspended labels and  $\mathcal{E}$  is the *data-state*. We have separated the *control-state* represented by  $\alpha$  and  $\gamma$  from the data-state  $\mathcal{E}$ , which maps values to variables. The active set  $\alpha$  represents the current values of the program counters. A subset of the operational rules that update the state configuration is given in Table 2.1. The rules determine how the statements update the state of a system and we give a short explanation of the axioms next.

The **par** statement can act when its label is active; it adds all of its arguments to the active set. Note that the **par** statement just creates new threads. It is the job of the policy to decide how to schedule the threads. The **goto** statement is used to switch from an active state to another state. It can take many labels as parameters, which allows it to be used to model fork transitions. A **goto** is enabled when its label is active; it removes the subtree containing the source and target labels from the active set (given by the *subtree* helper function), and activates the target labels (belonging to state statements) along with any labels of **trap** statements or entry actions that are ancestors of the target labels (the *entry-stats* helper function). The **trap** statement is defined by two rules representing the cases where the guard evaluates to true or false. The *trap\_1* rule shows the true branch, which adds the label of the **do** statement to the active set, executes it immediately (atomically), and leaves the system in the state reached by this execution.

## 2.3 Scheduling Semantics

The presented operational semantics leaves nondeterminism in the scheduler when several labels are active at once. This nondeterminism is resolved by the *scheduling policies*. An execution engine picks labels from the set of active labels  $\alpha$  according to the policy in the scope, and then executes the rule corresponding to the statement of that label.

A scheduling policy connected to a **state** block, schedules the sub-states according to a given algorithm. A program is executed by repeatedly running the policy of the topmost state in the hierarchy. The topmost policy will then schedule the states down in the hierarchy, which can have different policies assigned to them. The entity that calls the scheduling policy of the top-level state can be thought of as a global clock in the system. To define a scheduling policy we need a helper function. A label that belongs to a simple statement is *enabled* iff the premiss of the rule that corresponds to the statement holds; if a label belongs to a **state** statement, it is enabled iff it is active or has descendants that are enabled. A label can also become blocked by the scheduler, and is then not enabled. The  $\text{enabled}(l, \sigma, \beta)$  function returns the set of enabled labels that are descendants of  $l$ . We will use this function in the actual definition of the scheduling policies. A scheduling policy function accepts as parameters the current state of the program  $\sigma$  and the blocked set  $\beta$  and returns the next state of the program. As an example, the algorithm on the right implements a *step* scheduling policy that executes

```

run( $\sigma, \beta$ )
 $\rho := \text{enabled}(\text{self}, \sigma, \beta)$ 
for each  $l_i$  in  $\rho$  do
     $\sigma' := l_i.\text{RUN}(\sigma, \emptyset)$ 
return  $\sigma'$ 

```

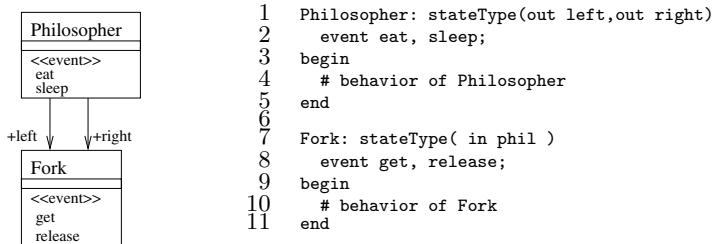


Figure 2.1. A UML class diagram and Rialto code

all enabled statements in the same step. Note that *self* refers to the **state** block whose *RUN* function was called. In the next section we will present other scheduling policies used to model different UML execution models. The scheduling policies have also been formalized as another level of operational rules working above the rules for the statements, but this work is not presented in this text.

### 3. Representing UML models in Rialto

In this section we describe how we translate a UML model into Rialto code. This transformation has to deal with the fact that UML is a family of languages, both at the semantic and the syntactic level. It is possible to create different translators for different semantic interpretations of UML. We currently support statecharts, activity diagrams and collaboration diagrams.

We show as an example the translation of a collaboration of objects whose behavior is modeled using statecharts or activity diagrams. Figure 2.1 shows a UML class diagram with two classes **Philosopher** and **Fork** along with the Rialto translation consisting of two corresponding **stateTypes**. There are two associations connecting the two classes, indicating that a **Philosopher** has a left and a right fork. This is reflected in the Rialto code by the two outgoing ports of the **Philosopher** type and the incoming one in the **Fork** type (we only use one incoming event queue for all connections). The behaviors of the two classes would be modeled using statecharts and the Rialto translations of the statecharts would appear inside the **stateTypes**.

#### 3.1 Statecharts

A **state** in a statechart is represented by a **state** block; hierarchical compound states are naturally written as nested **state** blocks. The first **state** block in a compound state is the **initial state**. **Transi-**

**tions** can be represented using the **trap** statements for monitoring an event or value of a guard, and the **goto** statement to take the transition. Also **orthogonal regions** are represented by **state** blocks, with a **par** statement specifying that the states (regions) are to be run orthogonally. **Fork** pseudostates are simply achieved by providing the **goto** statement with several state labels as parameters (**goto(a,b)** does a transition to the two states **a** and **b**, which are run orthogonally). **History pseudostates** are represented by **suspend** statements, that store the state of a block until it is re-entered.

The semantics of UML statecharts is based on a run-to-completion (RTC) step. The RTC algorithm fires transitions in the statechart based on the event on the queue, and dispatches it at the end of the RTC step. In addition to the translations described above, we need to provide a scheduling policy that runs the code according to the RTC algorithm. A state executing the RTC policy runs until no more runnable labels exists in its scope. The algorithm is shown below. It first runs all the enabled statements, and then checks if there appeared any new enabled labels, adding any labels that are not ancestors of the original enabled labels to the blocked set (line 5). If there are enabled non-blocked labels, it runs again. By this blocking, we ensure that when a state transition is taken during a RTC step, the newly activated state will not be executed during that same step.

```

run( $\sigma, \beta$ )
   $\rho := \text{enabled}(\text{self}, \sigma, \beta)$ 
  for each  $l_i$  in  $\rho$  do
     $\sigma := l_i.RUN(\sigma, \emptyset)$ 
     $\rho' := \text{enabled}(\sigma', \text{self}, \rho)$ 
     $\beta' := \{l_j \in \rho' \mid \mathcal{L}[l_j] = \text{state} \wedge l_j \notin (\text{ran}(\rho \triangleleft \downarrow^*))\}$ 
    if  $\rho' - \beta' \neq \emptyset$  then
      return  $\text{self}.RUN(\sigma', \beta')$ 
    dequeue;
  return  $\sigma$ 

```

The  $\triangleleft$  symbol is the domain restriction operator from the Z notation. In Figure 2.2 a) we show a simple statechart with a composite state divided into two orthogonal regions. The corresponding Rialto code is shown in Figure 2.2 b). This code has been generated by a tool as described in Section 2.4. The scheduling policy for the top-level **state** block in the code is set to RTC in line 2, and the queue **q1**, and the events are declared in line 3. The **S** state listens to event **e2** on queue **q1** and fires the transition to state **d** if it is present (line 5). The **par** statement in line 7 activates the regions **S1** and **S2** orthogonally.

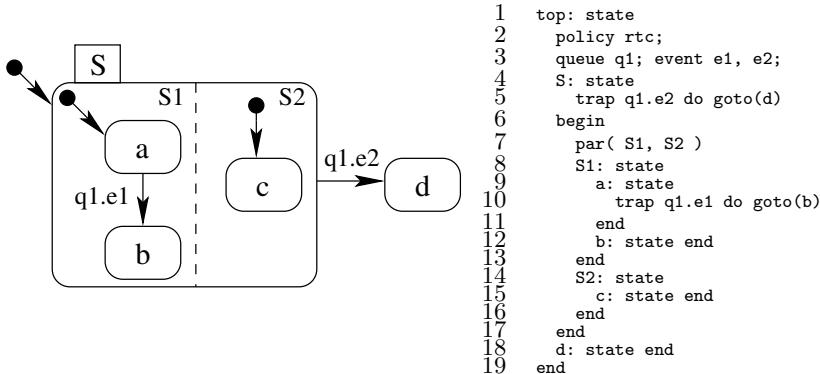


Figure 2.2. a) An example UML statechart b) the corresponding Rialto code generated by the tool

### 3.2 Activity Diagrams

The translation of activity diagrams follows a similar schema. The **activity states** are represented by **state** blocks, **forks** are dealt with as in the statecharts; however, **join** pseudostates usually need to be explicitly modeled using a **state** block for synchronizing the threads. The scheduling policy for an activity graph depends on the desired behavior, which is not specified in the UML standard. If we apply the RTC policy, the model will run in steps of one activity, i.e. at each invocation the active states will run to completion and the step will finish. Another possibility could be to specify a policy that runs the complete algorithm modeled by the activity graph at each invocation. Here we use the simple *step* policy introduced previously, which for each invocation runs all the enabled labels and returns (runs one statement in each thread).

As an example, we show a simple activity diagram and its translation into Rialto in Figure 2.3. After the activity in **a** is completed, we take a forked transition to **b** and **c** (line 6). This transition (`goto(b,c)`) activates both **b** and **c** that is they will run concurrently without using the `par` statement. The **join** state encapsulates the threads **b** and **c** and works as a synchronization boundary. It is sequentially composed with the **d** state, which it activates when all its children have finished.

### 3.3 Collaboration Diagrams

Statecharts and activity diagrams model the behavior of objects, while collaborations show an interaction between objects. We can use Rialto to capture the behavior of systems modeled using collaboration diagrams,

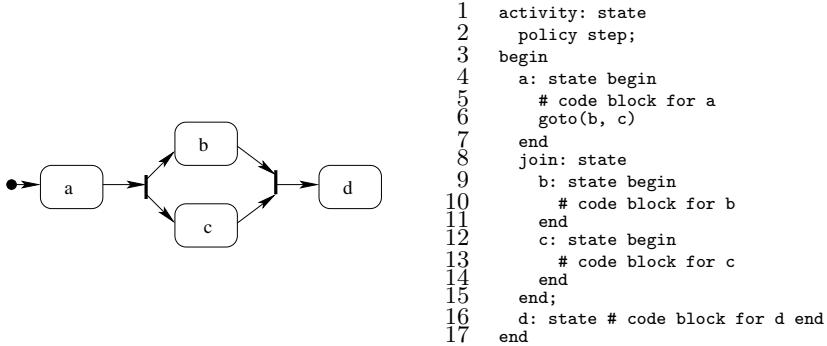
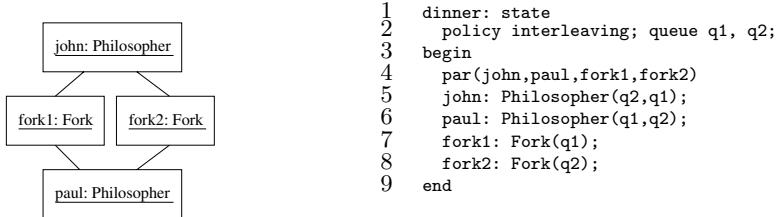


Figure 2.3. A simple Activity Diagram

where the objects can have behavior described by either statecharts or activity diagrams. The objects are represented by `state` blocks (or instantiations of `stateTypes`), which are defined to run concurrently using the `par` statement. The objects can communicate through queues. The default scheduling policy for a collaboration diagram is interleaving. This means that each object in the collaboration runs in its own task and is scheduled nondeterministically. The interleaving policy picks a label from the enabled set nondeterministically under the fairness assumption (The `:<` in line 2 in the algorithm on the right denotes nondeterministic assignment). Alternatively, the step policy can be used, which allows for active objects to execute at the same time during one step. When moving to code synthesis, we could have different physical mappings of the scheduling policies. For instance choosing a `interleaving_C_posix` policy would map the threads scheduled by the interleaving policy into posix threads instead of using the Rialto scheduler.

The behavior of an object can be described using a statechart or an activity diagram. The internal behavior of that object will then be RTC or step. As an example, we can model a collaboration of two philosophers and two forks as in Figure 2.4. The ports are connected to event queues when the types are instantiated. The associations from the class diagram are also instantiated as links connecting the objects. In the corresponding Rialto code, we show how the `stateTypes` are instantiated. The `par` statement activates all the objects in parallel. Two queues are declared in the top-level `state` block `dinner`. They are connected to the ports of the objects. This actually means that queue `q1` is the event

$$\begin{aligned}
& \mathbf{run}(\sigma, \beta) \\
& \rho := \mathbf{enabled}(l, \sigma, \beta) \\
& l_i :< \rho \\
& \mathbf{return} l_i.RUN(\sigma, \emptyset)
\end{aligned}$$



```

1 dinner: state
2   policy interleaving; queue q1, q2;
3 begin
4   par(john,paul,fork1,fork2)
5   john: Philosopher(q2,q1);
6   paul: Philosopher(q1,q2);
7   fork1: Fork(q1);
8   fork2: Fork(q2);
9 end
  
```

Figure 2.4. Objects and links instantiated

queue of the statechart of `fork1` while `q2` is the queue of the `fork2` object. The Rialto program is run by repeatedly calling the top-level scheduling policy. In this case it is the `interleaving` policy of the `dinner` state. That policy will pick one of the parallel blocks and call its scheduling policy, which in this model will be a `rtc` policy.

### 3.4 Automatic UML to Rialto Translation

The translation from UML models to Rialto is performed with the help of the SMW toolkit [10]. This toolkit is based on the Python programming language and it can be used to transform and extract information from UML models. It can read XMI [13] files created by any XMI-compliant UML editor. Once a model is loaded, we can navigate it using queries like in OCL and transform it using Python imperative statements.

In the code below, we use several functions for text manipulation: The vertical bars add a string to the final Rialto code. E.g. the statement `|2+2=|2+2` will add the string `"2+2=4"` to the output code. The indentation of the code is controlled using `|-->|` and `|<<-|`. The `*` operator on a text list generates a string with the concatenation of a list using the second operand as a separator. For example `["a", "b", "c"] * " or "` returns `a or b or c`.

The following function converts a UML state to Rialto. We assume that the reader is familiar with the OCL language and the UML metamodel for statecharts. The function accepts as parameters an object representing a state in a statechart and the name of the input queue to be used. We use the `trap` statement to model the high-level transitions that exit the state. The function is called recursively in the case of composite states that contain other states. In concurrent composite states, the subvertices are orthogonal regions that run in parallel using the `par` statement.

```

def StateVertex2Rialto(s,queue):
    s.name|: state|
  
```

```

|-->|
for t in s.outgoing:
    |trap| queue |.| t.trigger.getDescription() | do | Transition2Rialto(t)
|<-|
if s.oclIsKindOf(CompositeState):
    if s.subvertex.size()>0:
        |begin||-->|
        if s.isConcurrent:
            |par(| s.subvertex.name * "," |)|
            for subState in sortByInitialState(s.subvertex):
                StateVertex2Rialto(subState,queue)
        |<-||end|

```

We omit the implementation of the helper functions due to space constraints. The full script is available from the authors.

## 4. Animation and Code generation

One of the objectives of the Rialto language is to facilitate the animation, code generation and verification of UML models containing complex behavior. Using Rialto and our code generation approach, we can model the system using object-oriented techniques and the UML, while being able to generate efficient code in target languages that may not be object-oriented. This makes our method useful for designing embedded systems that can run on simple processors with limited memory and limited support for high-level programming languages. We can also generate hardware descriptions in VHDL for instance.

We currently have two strategies for generating code from Rialto. The simplest one, which we have implemented in C++, is an execution engine approach. The code generated using this approach is not optimized, but reflects the structure of the Rialto model, i.e. the generated code has one object for each Rialto statement. Therefore, this approach is suitable for animation. We will not explain this approach any further in this text due to the space limitations. The second strategy for code generation involves translating the model into finite state machines which are reduced using S-GRAPHS.

The optimized code generation process translates the Rialto program into a FSM representation, which can be reduced in a way similar to that of the POLIS approach [14]. This method is based on Software Graphs or S-GRAPHS and it is described in more detail in [11] and [15]. An S-GRAPH is a directed acyclic graph used to describe a decision tree with assignments, which can be reduced. From the obtained optimized low-level representation, we can easily generate target language code in e.g. C or VHDL. Compact assembly code could also be synthesized directly for targets lacking higher level language compilers.

The translation of a Rialto model into optimized code proceeds in five steps: Translation (flattening) of the Rialto code to a simple finite state machine, translation of the FSM into an S-GRAPH, optimization of the

S-GRAFH, translation of the S-GRAFH into a target language and, finally, compilation into machine code or synthesis of a hardware netlist.

We have achieved up to 30 percent code footprint reduction when generating C code and compiling it into object code. We expect also good results in area reduction when synthesizing hardware from VHDL code generated from Rialto. The Rialto compiler still needs development and more real-life examples need to be tried to get a better picture of the benefits.

## 5. Conclusions

We have shown how we can use Rialto to describe the combined behavior of different UML diagrams. Rialto can be converted into program code with the purpose of model animation or as a final production code to implement embedded systems with limited computational resources.

The UML behavior diagrams include many concepts such as actions, events, states, etc. that are not present in most popular programming languages, like C++ or Java. This means there is not a one-to-one mapping between a behavioral diagram and its implementation. Some model elements, like history states, can be implemented in many different ways; this clearly contrasts with class diagrams, that often can be easily implemented in a programming language supporting concepts like classes and objects, composition and inheritance. Rialto can be used as an intermediate language between models and code and supports semantic variations thanks to our two-phase approach for code generation.

An important decision in a code generation method is whether the programmer will be allowed to edit the produced code or not. We have opted to hide the final implementation from the programmer. This implies that the code does not need to be intelligible by a human programmer, and that it is not necessary to reverse engineer the code back into a UML model. However, this approach requires, in order to be practical, that the produced code is so efficient that the programmer does not need to tweak it by hand.

**Acknowledgements:** Dag Björklund gratefully acknowledges financial support for this work from the Nokia foundation.

## References

- [1] OMG, Unified Modeling Language Specification, Version 1.4, September 2001, available at [www.omg.org](http://www.omg.org)
- [2] S. Kent, A. Evans and B. Rumpe (Eds.) UML Semantics FAQ Available at [www.univ-pau.fr/OOPSLA99/samplewr99.eps](http://www.univ-pau.fr/OOPSLA99/samplewr99.eps)

- [3] Sabine Kuske, A Formal Semantics of UML State Machines Based on Structured Graph Transformations, UML2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Martin Gogolla and Cris Kobryn (Eds.)
- [4] Michael von der Beeck, Formalization of UML-Statecharts, UML2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Martin Gogolla and Cris Kobryn (Eds.)
- [5] Rik Eshuis and Roel Wieringa, An Execution Algorithm for UML Activity Graphs, UML2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Martin Gogolla and Cris Kobryn (Eds.)
- [6] Gérard Berry and Georges Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation, Science of Computer Programming, 19/2, 1992
- [7] D. Harel and A. Naamad, The STATEMATE semantics of statecharts, ACM Tran. of Software Engineering and Methodology, 5(4), 1996
- [8] Edward A. Lee, Embedded Software, Advances in Computers, 56, 2002
- [9] Stephen J Melloer and Marc J Balcer, Executable UML, A Foundation for Model-Driven Architecture, Addison-Wesley, 2002
- [10] Ivan Porres, A Toolkit for Manipulating UML Models, Technical report, Turku Centre for Computer Science, 441,2002
- [11] Dag Björklund, Johan Lilius and Ivan Porres, Towards Efficient Code Synthesis from Statecharts, pUML Workshop at UML2001,
- [12] Dag Björklund and Johan Lilius, A Language for Multiple Models of Computation, Symposium on Hardware/Software Codesign 2002, ACM
- [13] OMG, XML Metadata Interchange Specification, Available at [www.omg.org](http://www.omg.org)
- [14] Felice Balarin et al., Hardware-Software Co-Design of Embedded Systems, Kluwer Academic Publishers, 1997
- [15] Dag Björklund and Johan Lilius, From UML Behavioral Models to Efficient Synthesizable VHDL, Proceedings of the 20th IEEE Norchip Conference, 2002

## Chapter 3

# PLATFORM-INDEPENDENT DESIGN FOR EMBEDDED REAL-TIME SYSTEMS \*

Jinfeng Huang<sup>1</sup>, Jeroen P.M. Voeten<sup>1</sup>, Andre Ventevogel<sup>2</sup>, Leo van Bokhoven<sup>3</sup>

<sup>1</sup>*Faculty of Electrical Engineering Eindhoven Univ. of Tech., The Netherlands*

<sup>2</sup>*TNO Industrial Technology, The Netherlands*

<sup>3</sup>*Magma Design Automation B.V., The Netherlands*

**Abstract** With the increasing complexity of the emerging embedded real-time systems, traditional design approaches can not provide sufficient support for the development of these systems anymore. They especially lack the ability to trace and analyse real-time system properties. In this paper, we investigate the design difficulties for embedded real-time systems and propose several principles for coping with these difficulties, which should be incorporated by an “adequate” design approach. Several prevailing design approaches are evaluated against these principles and their merits and drawbacks are examined and illustrated by examples. Finally, a platform-independent approach (POOSL[8, 9] + Rotalumis[20]) is introduced to remedy these design problems for embedded real-time systems. Initial experiments have been performed that confirm the advantages of this approach.

**Keywords:** Platform-independent, Real-time, system design

\*This research is supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs, the Technology Foundation STW and the Netherlands Organisation for Applied Scientific Research TNO.

## 1. Introduction

Over the past decades, we have witnessed a significant increase of the application of embedded real-time (RT) systems. These applications range from microchip product lines with microsecond response time to electric appliance controllers with seconds or minutes response time. The common feature of these systems is that the correctness of the system depends not only on the value of the computation but also on the time when the results are produced [18]. To guarantee timeliness feature in the system implementation, we expect the system behaviour to be predictable and we would like to ensure in advance that all critical timing constraints are met. However, with the increase of complexity, traditional design approaches are not well-suited for developing time-critical applications. We list below several typical characteristics in traditional design approaches that lead to unpredictable behaviour of embedded RT systems.

**Craft-based system implementation.** Traditional methods are often bottom-up and driven by the specifics of the implementation platform. Critical timing constraints are dealt with in ad-hoc heuristic fashions. During system development, developers make implementation choices based on rough estimations of computation times. However, these computation times are in fact influenced by many non-deterministic factors of the underlying platform<sup>1</sup>. This severely hampers to make adequate estimations of computation times. Hence inadequate estimations can easily result in faulty design and implementation decisions.

**Ineffective design languages.** Traditional programming or modelling languages usually do not provide sufficient support for embedded RT systems.

- 1 These developing languages usually have platform-dependent execution semantics, which means that the behaviour of the system is heavily affected by the (non-deterministic characteristics of the) underlying platform. This makes it very difficult to establish design correctness.
- 2 Design languages often have inadequate support for modularisation. Modularity means that the system can be decomposed into independent subsystems and the composition of these subsystems have little impact on their own properties and functionalities[15]. Object-orientation helps developers improve the modularity of the software by grouping together data-structures and operations and encapsulating them from the outside world. In this way, a com-

plex system can be divided into independent subsystems that can more easily be designed, analysed and composed. However, in embedded RT systems, module independency is ruined by the fact that all modules running on one processor share the time resource. Therefore, after the composition of subsystems, the original real-time properties of the individual subsystems can not be sustained.

- 3 Last but not the least, design languages often have little or no support for expressing timing requirements (such as deadlines and periods). This means that timing requirements cannot be taken into account when the system is mapped onto the target platform.

Due to the above pitfalls, embedded real-time systems lack portability, adaptability, maintainability and reusability. Furthermore, the correctness and performance has to be established by extensive testing on the target platform. The disadvantages are obvious:

- Tests are usually carried out at the observable boundary of the system. It is hard to locate the specific internal details that cause (timeliness) errors.
- Additional test code may change the (real-time) behaviour of the system and may cause new errors (Heisenberg principle in testing [24, 16]).
- Test can only be carried out at a late stage in the design cycle. Failures detected at this stage often lead to expensive and time-consuming design iterations.
- Test results are affected by many uncontrollable external (environmental) and internal (non-deterministic factors). Some “rare” errors are hard to capture or repeat by test.

## 2. The Dream: Platform-Independent Design

The key problem of traditional approaches mentioned in the previous section is that low-level details have a big influence on the system behaviour as a whole. Traditional design approaches do not provide facilities to abstract from these details adequately. In this sense, embedded RT system design suffers from “butterfly-effects”: a cache miss might result in a missed deadline. To cope with this problem, design approaches should offer adequate abstraction facilities and shield the design from the details of the platform. Platform-independent approaches [12, 17] have been proposed to address this problem, which ideally have the following characteristics:

**A well-founded<sup>2</sup> and expressive modelling language.** The core of such a design approach is a modelling language which can help designers express and verify their design ideas in an adequate way. This means that the language should be expressive, should have platform-independent semantics, operational semantics and adequate support for modularization:

- 1 *Adequate expressive power:* Embedded RT systems often have features of timeliness, concurrency, distribution and complex functionality. To be able to express these features in a model, a modelling language should have facilities to describe system structure, concurrency, communication, data types, non-determinism and timing.
- 2 *Platform-independent semantics:* The non-deterministic factors introduced by the underlying platform make the simulation and verification results of the design model unreliable and unpredictable. Platform-independent semantics of the modelling language gives a unique interpretation of the model and makes simulation and verification platform independent. Furthermore, it provides the flexibility to reuse the design model and target it to different or modified platforms.
- 3 *Operational semantics:* Operational semantics of the modelling language lends itself naturally to executability. Inconsistency between different aspects can then be located and correctness and performance properties can be checked either exhaustively (e.g. model-checking) or non-exhaustively (e.g. simulation). As a result, many design errors can be corrected in an early development stage, avoiding costly and time-consuming iterations.
- 4 *Modularity support:* Modularity is generally considered as the only available way for managing complexity of a system. Since complexity is a common feature of current embedded RT systems, it is necessary to embody modularity in the modelling language. This means that a concept of modules should be supported in such a way that module composition does not cause the real-time properties of the individual components to change. This allows components to be composed on basis of their interfaces, without having to understand the internal details.

**Automatic and correctness-preserving transformation.** System generation should take a design model as blueprint and build a complete hardware/software implementation. Preferable, the generation of the

		Rational Rose RealTime (ROOM)	Cinderella SDL (CSDL)	TAU Generation 2 (Tau2)	Esterel
<i>Design language</i>		UML	SDL <sup>*</sup>	SDL+UML <sup>**</sup>	Esterel
<i>Expressive Power</i>	<i>Timing</i>	limited	limited	limited	good
	<i>Structure, data type, concurrency, communication</i>	yes	yes	yes	yes
	<i>non-determinism</i>	yes	yes	yes	no
<i>Platform-independent Semantics</i>		no	no	yes	yes
<i>Operational Semantics</i>		yes	yes	yes	yes
<i>Modularity Support</i>		limited	limited	yes	yes

(a)

		Rational Rose RealTime (ROOM)	Cinderella SDL (CSDL)	(TAU2) TAU Generation 2	Esterel
<i>Target language</i>		C, C++, Java	no	C, C++	C, C++, Java, Hardware
<i>Automatic generation</i>		yes	no	yes	yes
<i>Correctness-preserving transformation</i>		no	no	no	yes

(b)

Figure 3.1. Comparison of several design approaches – The semantics of SDL is based on SDL-96 in CSDL[1]. \*\* TAU2 integrates concepts of SDL-2000 and UML 2.0 in its modelling language[3].

implementation should be largely automated. Furthermore, the transformation should also guarantee that the implementation behaves the same as the model. In this way, many errors caused by human-factors can be avoided during the system generation and correctness of the system can be guaranteed during the transformation.

The above characteristics of design approach can help designers overcome the existing drawbacks of traditional design approaches and serve as guidelines for new embedded RT system design approaches. In the sequel, we will discuss several prevailing design approaches for RT systems and evaluate them based on the above characteristics.

### 3. Comparison of several design approaches for embedded RT systems

Figure 3.1 gives a brief comparison of several typical design approaches<sup>3</sup>, all of which provide a powerful design language and most of which also support automatic software/hardware generation. In this way, software/hardware productivity can be improved and costly design interpretation errors can be reduced. However, none of them satisfies all those

characteristics of platform-independent design. In the following, we will investigate how those characteristics are supported by these approaches.

### 3.1 Expressive power

**Timing.** CSDL and TAU2 only support time delays. There is no explicit timing expression in Esterel. Instead, it captures the time passing by counting activations of the system [5]. The modelling of physical time can be accomplished by counting activations issued at a regular time interval. Due to the deterministic characteristic of Esterel models (see below in this section), this form of real-time mechanism is adequate enough for describing and analysing timing constraints. Compared to the other three approaches, ROOM [2] provides various timing services to express different timing constraints, such as delay timers, periodic timers, informIn timers. However, its platform-dependent semantics does not give an unambiguous interpretation of time expressions in the model, which inhibits designers from analysing and predicting timing behaviour of the model.

**Structure, data type, concurrency and communication.** These features are supported by all of these approaches. In this paper we are not going into detail about how these approaches support these features. For more information, we refer to [23, 19].

**Non-determinism.** Non-determinism is an essential way to manage the system complexity. It not only leaves freedom to obtain optimal implementations but also largely reduces the complexity of the model by abstracting it from irrelevant implementation details. Except for Esterel, all the three approaches can describe an embedded RT system at a high level of abstraction by using non-determinism. At the same time, they offer the possibility to model interested details of the system. Esterel sacrifices non-determinism to obtain a deterministic model of the system, in which the system behaviour is predictable. The cost of the deterministic characteristics is that complex behaviour is difficult to be modelled and analysed efficiently.

### 3.2 Platform-independent semantics

CSDL relies on an asynchronous timer mechanism which is able to access a global clock referring to the physical clock<sup>4</sup>. In this time mechanism, the delay between the moment of timer expiration and the moment at which the model reacts to this expiry is unbounded [23]. The unbounded delay is caused by several factors, such as the waiting time of

timer-expired message before it is inserted into the input message queue of the process, the time for consuming all messages before the timer message and the interaction time between the process and its input message queue [13]. When a timer is set to  $t$  seconds, the interpretation of this timer is in fact an arbitrary time duration  $d_t$  ( $d_t \in [t, \infty)$ ). Such a weak interpretation of timers cannot provide enough expressive power to describe the timing behaviour of real-time systems. The unbounded uncertainty of  $d_t$  contradicts with the predictability of real-time system behaviour.

**EXAMPLE 3.1** *Consider a simple digital clock, which issues an action at the end of each second to count the time passing. Due to the unbounded uncertainty of a timer in CSDL, it is impossible to ensure that actions can be issued at (or close to) the expected time points.*

The time mechanism in CSDL is heavily affected by the platform-dependent physical clock. Such a platform-dependent timing mechanism cannot provide facilities to debug and analyse timing behaviour of a model, because any debugging and analysis observation may introduce extra time passing which changes the real-time behaviour of a model and leads to unreliable debugging and analysis results. The same problem holds for the ROOM approach.

Different from CSDL and ROOM, TAU2 adopts a two-step execution model[14], the state of a system can change either by asynchronously executing some atomic actions such as communication and data computation without time passing (phase 1) or by letting time pass synchronously without any action being performed (phase 2). The semantics of Esterel is based on the perfect synchrony hypothesis[5], which assumes that a system's reaction to an input is instantaneous. The time measurement is achieved by counting the number of events. The computation models of both TAU2 and Esterel adopt a virtual time, which is physical-time independent. In this way, real-time behaviour of their models is always predictable with respect to this virtual time. Therefore, the above unbounded uncertainty problem does not exist in these design approaches. Furthermore, in these approaches, a model can be uniquely interpreted and analysed by verification and simulation techniques. Based on the analysis results, the model can be refined to meet predefined timing requirements. A clock example in section 4.1 illustrates that timing behaviour can be adequately modelled by such design approaches.

### 3.3 Modularity support

All of these approaches have object-oriented characteristic supporting data and functionality encapsulation. Due to the platform-dependent

semantics of CSDL and ROOM, timing characteristics of every module in the model are disturbed by the time consumption of other modules in the model and by other processes running on the same platform. On the contrary, both Esterel and TAU2 assume the underlying platform to be infinitely fast and the timing behaviour of the system is not constrained by the computation power of the underlying platform. Therefore, the composition of separate modules does not change their timeliness. As a consequence, the analysis of the design is easier and predictable.

### 3.4 Correctness-preserving transformation

As shown in Figure 3.1b, all approaches facilitate automatic software or hardware generation except for CSDL. In ROOM, automatic code generation is accomplished by linking a model to a so-called service library which act as a virtual machine on top of different target platforms. In this sense, the implementation is in fact an executable model and problems encountered in the model, such as platform-dependent semantics, are automatically inherited in the implementation. Although TAU2 can provide a reliable way to analyse a model and refine it to ensure the property correctness, it does not have a transformation mechanism to guarantee that correctness properties verified in the model can be transferred to the implementation. In the automatically generated implementation, timing expressions are simply interpreted as unbounded physical time, which faces the same unbounded uncertainty problem as in the CSDL model. The issuing time of actions can deviate much from those observed in the model. Furthermore, the ordering of events can also be different from those observed in the model. Here are several examples.

#### EXAMPLE 3.2 *Accumulated timing errors:*

*Consider a digital clock whose functionality is similar to that described in Example 1. The difference is that its accuracy is one tenth second instead of 1 second. Figure 3.2 shows that the timing errors<sup>5</sup> is accumulated during the execution of the implementation, which is automatically generated from the clock model in TAU2.*

#### EXAMPLE 3.3 *Incorrect functionality caused by accumulated timing errors:*

*Consider a controller for a flash board showing 4 consecutive letters “IEEE”, whose functionality is described as follows. The four letters of the word are sequentially displayed on the board, and then wiped off altogether at the same time. The iteration will continue unless being interrupted manually. One solution to designing this controller is to use three parallel processes. Process **I** emits letter I every 0.3 seconds, process **E** emits letter E every 0.1 seconds and process **space** issues four*

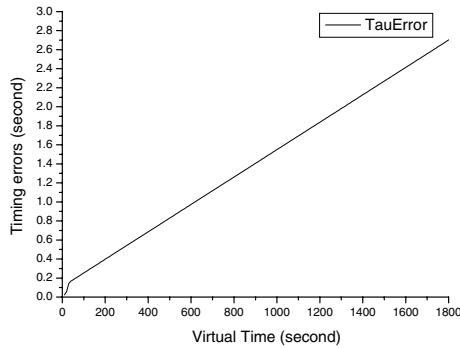


Figure 3.2. Accumulated timing errors of a TAU2 clock implementation



Figure 3.3. Output of the controller

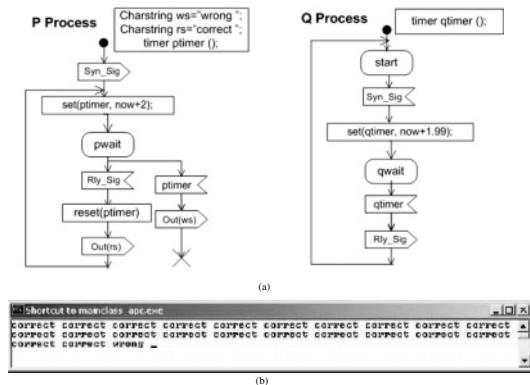


Figure 3.4. Two parallel processes and the output of their TAU2 implementation

blank spaces every 0.3 seconds to erase the letters. The three processes starts from 0.01, 0.02 and 0.25 second respectively. It is easy to verify that the model behaves correctly according the functionality specification in the tool of TAU2. However, we will see a different picture when we look at the behaviour of the implementation (working with the physical time). Figure 3.3 gives a snapshot of the output of the implementation. We can see that after several iterations, the accumulated timing errors has led to an incorrect output sequence.

**EXAMPLE 3.4 Incorrect functionality caused by duration actions:** Not only can accumulated timing errors lead to incorrect event order, computation expense of individual actions can also result in unexpected behaviour. Consider the simple example of Figure 3.4a. Two parallel processes  $P$  and  $Q$  synchronize with each other at the beginning of each iteration to avoid the accumulation of timing errors.  $P$  sets a timer with 3 seconds delay and  $Q$  sets a timer with 2.99 seconds delay. After the timer of  $Q$  expires,  $Q$  sends a “reply-signal” message to  $P$ . At the  $P$  side, there are two possibilities:

- 1  $P$  receives the timer expiration message and outputs the message “wrong”.
- 2  $P$  receives the reply message from  $Q$ , resets its own timer and outputs the message “correct”.

It is not difficult to verify that the output message of the  $P$  should always be “correct” in the model. However, the automatically generated software implementation shows an unexpected behaviour (see Figure 3.4b).

Different from TAU2, Esterel provides a correctness-preserving transformation from a model to an implementation. The Esterel language is a synchronous concurrent programming language based on perfect synchrony hypothesis, which assumes computation actions take zero time. A pure Esterel model can be implemented in a digital circuit or software by using a formal transformation. Such a transformation keeps the semantics of the Esterel model to the hardware/software implementation except that perfect synchrony hypothesis is replaced by digital synchrony hypothesis (i.e. zero time is replaced by one cycle clock). The correctness of the generated implementation relies on the fact that perfect synchrony does not deviate very much from digital circuit synchrony [5].

## 4. Towards Platform-independent Design

In the previous section, we have reviewed several typical design approaches for embedded RT systems and analysed the drawbacks and

merits of each approach. Among them, TAU2 provides a relatively good support for modelling complex embedded RT systems but it provides no facilities for correctness-preserving transformation from a model to its implementation. Esterel has full support for correctness-preserving transformation for automatic hardware/software generation but its modelling language lacks ability to model complex interactive RT software systems[6]. In this section we propose an approach which considers all aspects introduced in section 2 and aims at platform-independent design. It provides an expressive and well-founded language (POOSL) for modelling complex embedded RT systems and a correctness-preserving transformation tool (Rotalumis) for automatic software generation. Its ability to preserve correctness during transformation have been proven in [10].

## 4.1 POOSL

In this section we give a brief overview of the POOSL language (Parallel Object-Oriented Specification Language), which is employed in the SHEsim tool and developed at the Eindhoven University of Technology. The POOSL language is equipped with a complete mathematical semantics and can formally describe concurrency, distribution, communication, real-time and complex functionality features of a system in a single model.

Similar to TAU2, the semantics of the POOSL language is based on a two-phase execution model which guarantees a unique interpretation of the model. Hence, behaviour of the model is not affected by underlying platforms. The detailed mathematical framework behind the POOSL language is given in [8, 20], and a formal description of its execution engine can be found in [9]. Figure 3.5 shows a simple clock in POOSL which performs the same functionality as in Example 1. Each event is accurately issued at the expected (virtual) time in this model (note that the  $i$ th event is issued at the  $i$ th second.).

Because of the expressiveness and well-founded semantics of the POOSL language, it has been successfully applied to model and analyse many industrial systems such as a network processor[19] an microchip manufacturer device[11] and a multimedia application[22].

## 4.2 Rotalumis

In this section, we outlines the formal transformation mechanism of software generation tool Rotalumis, which can transform a POOSL model into a software implementation for single processor platforms [21]. Different from other software generation tools, the Rotalumis supports

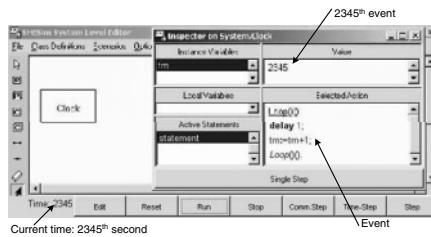


Figure 3.5. A clock model in SHEsim

the correctness-preservation during the transformation by adopting the following techniques.

- 1 Execution trees are used to bridge the gap between the expressibility difference between POOSL and C++ language. POOSL provides ample facilities to describe system characteristics such as parallelism, preemption, nondeterministic choice, delay and communication that are not directly supported by C++. In order to provide a correct and smooth mapping, execution trees are adopted to represent individual processes of the model and a scheduler calculates their next step actions in the execution of the model. The correctness of this execution method with respect to the semantics of the POOSL model has been proven in [9]. Therefore, the generated C++ software implementation will always have the same (untimed) event order as observed in the POOSL model. More details about the execution trees can be found in [20].
- 2 Correctness property preservation is guaranteed by the  $\epsilon$ -hypothesis in the software implementation, which assumes that the timed execution trace of the implementation is always  $\epsilon$ -neighbouring<sup>6</sup> to a timed execution trace in the model. It has been proven that the evolvement of execution trees always follows one of the state traces in the model. Furthermore, during the execution, the scheduler of execution trees tries to synchronize the virtual time and the physical time, which ensures that the execution of the implementation is always as close as possible to a trace in the model with regard to the distance between timed state sequences. Due to the limitation of the platform, the scheduler may fail to guarantee the timing constraints specified in the model, even with  $\epsilon$ -neighbouring relaxation. In this case, designers can get the information about the missed actions. Correspondingly, they can either refine the model

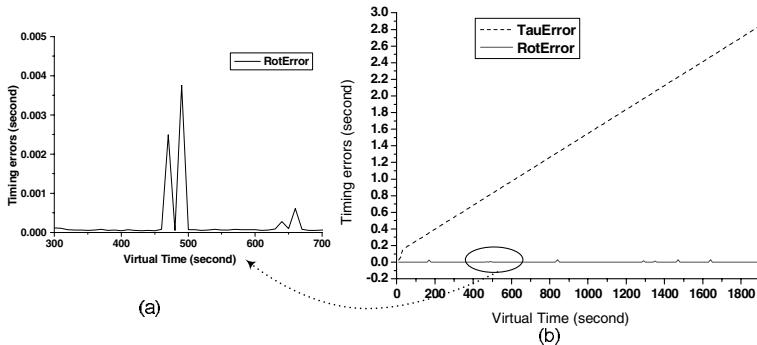


Figure 3.6. A comparison of the timing errors

and reduce computation cost, or replace the target platform with a platform of better performance.

In Figure 3.6b, we give a comparison of the timing errors of two automatically generated clocks from TAU2 and Rotalumis respectively, which run on the same platform<sup>7</sup>. The functionality of the clocks is as stated in Example 2. The clock generated by Rotalumis is supposed to satisfy 0.1-hypothesis (i.e, the scheduler tries to synchronize the virtual time and the physical time within 0.1 second difference.). Figure 3.6a shows that the timing errors of the implementation is controlled within 0.1 second and does not accumulate with the time passing<sup>8</sup>. In addition to the clock example, errors in Example 3 and Example 4 can also be avoided in the generated implementations by Rotalumis.

## 5. Conclusions

In this paper, we analysed the difficulties experienced when designing complex embedded RT systems and proposed several essential characteristics that an “adequate” design approach should possess in order to overcome these difficulties. Several prevailing approaches have been evaluated based on how well they support these characteristics. The experiments on these approaches indicate that none of them has full support for complex embedded RT system design. A platform-independent design approach is proposed which considers all essential characteristics. The approach consists of two individual procedures, platform-independent design and correctness-preserving transformation. Platform-independent design guarantees an unambiguous interpretation to the design description, whose performance and correctness can be analysed by verification and simulation techniques. Based on the anal-

ysis results, a design description can be refined to meet the predefined requirements, which is a prerequisite for the procedure of the correctness-preserving transformation from the model to the implementation. The correctness-preserving transformation takes a model as input, and generates a complete and executable implementation whose correctness is guaranteed during the transformation. The time and cost to perform test is thereby saved. The tools for both procedures have been developed at the Eindhoven University of Technology. SHEsim tool with POOSL language provides a platform independent environment for designing, simulating and analysing a model. Rotalumis tool can automatically transform a POOSL model into executable code for target platforms preserving the correctness properties verified in the model. Initial experiments have been performed that confirm the advantages of this approach.

## Notes

1. A platform, in this context, refers to those hardware and software infrastructures on which the implementation relies. Typically, a platform includes hardware, operating systems and middle-ware as well. Non-deterministic factors of the platform are caused by techniques which have been widely applied to boosting the overall computation performance, such as caches, pipelines, instruction pre-fetch techniques and memory management[7].
2. A well-founded language refers a language with formal semantics which enables an unambiguous interpretation on its expressions
3. Figure 3.1a gives a comparison of design languages of these approaches and Figure 3.1b gives a brief comparison of their code generation ability and quality.
4. Although SDL-2000 uses a virtual time to count the time passing in its models[4], SDL-96 adopts physical time[13].
5. Timing errors represent the deviation of the issuing time of actions in the implementation from that in the model, i.e., they represent difference between the virtual time and the physical time of the issuing actions.
6. A timed execution trace is a state sequence with a time interval attached to every state. If two timed execution traces are  $\epsilon$ -neighbouring, they have exactly the same state sequence and the least upper bound of the absolute difference between the left-end points of the corresponding intervals is less than or equal to  $\epsilon$ . For more information, see [10].
7. CPU 300Mhz, Memory 128M and Windows 2000
8. Several peaks in Figure 3.6a are caused by the underlying OS (windows 2000). We have tried to execute the same implementation in other OS, such as windows 98, and no such high peaks exist. In most situations, timing errors are around  $5 * 10^{-5}$  seconds in Rotalumis. Note that the timing errors of both implementations are observed after the issuing of the actions, i.e. the actual timing errors should be less than those shown in Figure 3.6.

## References

- [1] Cinderella SDL 1.3. <http://www.cinderella.dk/>.
- [2] Rational Rose RealTime. <http://www.rational.com/tryit/roser/index.jsp>.
- [3] TAU Generation 2. <http://www.taug2.com/>.

- [4] Z.100 Annex F1: Formal Description Techniques (FDT)– Specification and Description Language (SDL). Telecommunication standardization sector of ITU, Nov 2000.
- [5] G. Berry. A hardware implementation of pure esterel. In *Academy Proceedings in Engineering Sciences*, volume 17, pages 95–130. Indian Academy of Sciences, 1992.
- [6] G. Berry. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel, pages 425–454. MIT Press, 2000.
- [7] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.
- [8] P.H.A.van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1997.
- [9] M.C.W Geilen. *Formal Techniques for Verification of Complex Real-time Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2002.
- [10] Jinfeng Huang, Jeroen Voeten, and Marc Geilen. Real-time Property Preservation in Approximations of Timed Systems. In *Proceedings of First ACM & IEEE International Conference on Formal Methods and Models for Codesign*, Mont Saint-Michel, France, June 2003. IEEE Computer Society Press.
- [11] Jinfeng Huang, Jeroen Voeten, Pieter van der Putten, Andre Ventevogel, Ron Niesten, and Wout van de Maaden. Performance evaluation of complex real-time systems, a case study. In *Proceedings of 3rd workshop on embedded systems*, pages 77–82, Utrecht, the Netherlands, 2002.
- [12] L. Lavagno, S. Dey, and R. Gupta. Specification, modeling and design tools for system-on-chip. In *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, pages 21–23, 2002.
- [13] S. Leue. Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification*, Chapman & Hall, 1995.
- [14] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. G. Larsen, editor, *Proceedings of the 3rd*

- workshop on Computer-Aided Verification, Alborg, Denmark, July 1991.
- [15] H. A. Partsch. *Specification and transformation of programs: a formal approach to software development*. Springer-Verlag New York, Inc., 1990.
  - [16] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture: 1st edition*. John Wiley & Sons, Inc., Oct 1996.
  - [17] A. Sintotski, D.K. Hammer, O. van Roosmalen, and J. Hooman. Formal platform-independent design of real-time systems. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 163–170, 2001.
  - [18] J. Stankovic and K. Ramamritham, editors. *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, 1988.
  - [19] B.D. Theelen and R.D.J. Kramer J.P.M. Voeten. Performance Modelling of a Network Processor using POOSL. *Journal of Computer Networks, Special Issue on Network Processors*, pages 667–684, April 2003.
  - [20] L.J. van Bokhoven. *Constructive Tool Design for Formal Languages from semantics to executing models*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2002.
  - [21] L.J. van Bokhoven, J.P.M. Voeten, and M.C.W. Geilen. Software Synthesis for System Level Design Using Process Execution Trees. In *Proceedings of 25th Euromicro Conference*, pages 463–467, Milan, Italy, 1999. IEEE Computer Society Press, Los Alamitos, California.
  - [22] F.N. van Wijk, J.P.M. Voeten, and A.J.W.M. ten Berg. An abstract modeling approach towards system-level design-space exploration. In *Proceedings of the Forum on specification and Design Language*, Marseille, France, September 2002.
  - [23] J.P.M. Voeten, P.H.A. van der Putten, M.C.W. Geilen, and M.P.J. Stevens. System Level Modelling for Hardware/Software Systems. In *Proceedings of EUROMICRO'98*, pages 154–161, Los Alamitos, California, 1998. IEEE Computer Society Press.
  - [24] H.P.E. Vranken. *Design for test and debug in hardware/software systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1998.

## Chapter 4

# REAL-TIME SYSTEM MODELING WITH ACCORD/UML METHODOLOGY

*Illustration through an automotive case study*

Trung Hieu Phan, Sebastien Gerard and Francois Terrier

{TrungHieu.Phan, Sebastien.Gerard, Francois.Terrier}@cea.fr,

**Abstract** Engineers are now facing difficult problems in developing ever more sophisticated real-time systems with reduced time-to-market constraints. Object-oriented modeling with UML provides useful answers to these challenges. However, UML specification of real-time features and implementation techniques for real-time applications still require improvement. The ACCORD/UML methodology outlined in this paper therefore proposes UML extensions for modeling real-time complex systems with a high level of abstraction, followed by a basic technology for prototyping and a set of validation methods. ACCORD/UML is fully in line with the MDA paradigms proposed by the OMG. It focuses on system real-time constraints specification and ensures automatic developing phases to guarantee safe construction and application correctness. Throughout this paper, an automotive case study, based on a car seat adjustment system, is used to illustrate the ins and outs of ACCORD/UML methodology.

**Keywords:** UML, Real-time constraints, modeling, validation

## 1. Introduction

This article begins “filling the blanks”, by providing a brief introduction to UML and other UML-based methods dedicated to the real-time domain. This includes an overview of their advantages and disadvantages to facilitate comparison with the ACCORD/UML method described later in the paper.

UML [1], which relies on object-oriented modeling with a standard formalism, combines reusability and evolutivity to systems development

and has now became a de facto standard for modeling. Despite its success, UML has not proven fully satisfactory in a real-time domain. While there are specific UML concepts suitable for modeling certain DRES [2] features, the language in its standard form does not meet all the requirements set by real time applications engineers. To overcome this hurdle, a dedicated UML profile, known as "Scheduling, Performance and Time" (SPT) [3] has been standardized. The SPT profile specifies a minimum number of concepts required to model the real-time aspects of any system. A model adorned with these additional concepts (i.e. new stereotypes and tagged values) can then be used for implementation support and schedulability-performance analysis. SPT has two essential components: a general resource management package and a real-time analysis package. These are devoted to defining both generic concepts required to model real-time features (i.e. mainly Resource and Quality of Service concepts), and UML extensions required to adorn models focused in schedulability-performance validation. OMG has also launched a UML profile for Quality of Service and Fault Tolerance[4]. This profile specifies extensions to model fault tolerance features in a UML model by defining a family of QoS and reusing the generic concepts defined in SPT.

A certain number of solutions are provided by other commercially available methods (and tools), including ARTiSAN [5], [6], [7], UML/SDL with ObjectGeode [8], [9], [10], [11], ROOM with ObjectTime (now known as UML-RT with Rose-RT [12], [13]) and RT-UML with Rhapsody [15], [14] , [16]. These methods generally propose two separate models: the conventional object-oriented model of application structure and the task architecture specification with a concurrency diagram. Another common point of these methods is that real time constraints are expressed in the low-level mechanism implementing them (most often specified via timers). Use of both orthogonal models requires the developer to manage in parallel these two conceptual points of view and requires a high degree of expertise in both real time development and object-oriented modeling and programming. Additionally, the way of declaring real-time constraint and real-time behaviors restricts the real independency between implementation choices and application modeling.

ACCORD/UML [17], [18], [19] tackles these problems by extending UML language to completely describe embedded and real-time characteristics. It not only guides end users throughout the development of their real-time applications but also serves as a formal base for the analysis and validation of functioning constraints. In this method, integration of the ACCORD kernel provides operational support for the execution

of real-time applications such as: inter-object communications and synchronizations, concurrency control of data access, global scheduling for message execution. This kernel is dedicated to rapid system prototyping and validating by simulation.

The following discussion is divided into six chapters. The first briefly describes a seat adjustment system that is used in the rest of the article to illustrate our approach. The next chapters cover different phases in the life cycle of the proposed method: Preliminary Analysis Modeling, Detailed Analysis Modeling and Validation. Finally, the advantages of our ACCORD/UML methodology are summarized, and currently ongoing research orientations are outlined in a conclusion.

## 2. Case study

This section describes the specifications set for an automotive seat control system by Daimler Chrysler [20]. This system then serves as a case study to illustrate use of the ACCORD/UML methodology. The system to be designed is intended to control of a premium car seat (seat movements, heating functions, etc.) as visually represented in figure 1.

The seat incorporates 28 sensors and 6 motors divided into 2 groups. Each group has three motors with three different levels of priority. Two motors in two different groups can be activated simultaneously. Within the same group of motors, only the higher priority motor is activated. Release of an appropriate button enables deactivation of the motor within 250ms. Each seat drive unit is fitted with a Hall sensor, which indicates adjustment axis motion by a corresponding number of ticks. The motor for the relevant direction of seat adjustment is switched off five ticks before stop position to avoid a short-circuit condition. When supplied with power for the first time, the seat is calibrated by moving all adjustment axes to their stops. The seat is stopped when the Hall sensor supplies no more ticks for 250 ms.

Storing the position of the seat entails first pressing a memory button, then, within two seconds, the desired storage button. Pressing the relevant storage button can restore the saved memory position. The motors are activated in the order of priority laid down for the groups. They are activated only for as long as the storage button is pressed. The "courtesy" seat adjustment is a longitudinal movement to a predefined rearward position that makes it easier for the driver to climb in or out.

The seat heating functions can be activated when circuit 15R or circuit 15 is enabled. The heating system has two heating stages. When heating stage 2 is switched on, the system remains activated for five minutes, after which the system automatically switches to stage 1. When stage 1 is

switched on, it remains activated for 30 minutes; and the heating system subsequently switched off. All functions of the system are deactivated on under-voltage or over-voltage detection. Seat adjustment is stopped at a road speed of more than 5 km/h. The heating system is temporarily deactivated when the seat adjustment function is used.

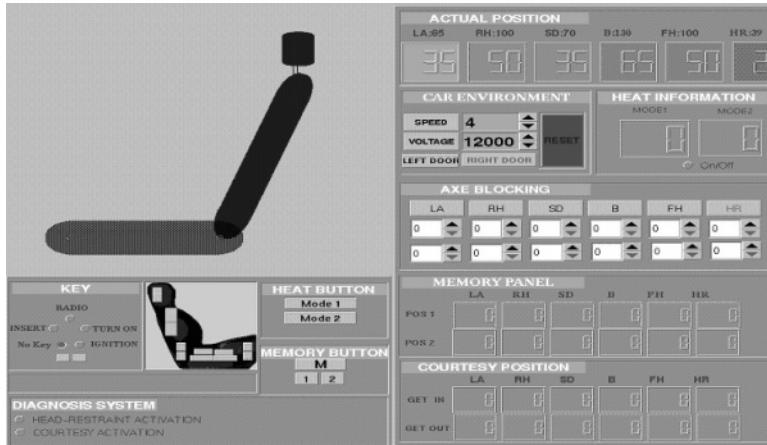


Figure 4.1. GUI of the seat control system developed for system simulation.

### 3. Preliminary Analysis Model (PAM)

ACCORD/UML methodology defines three successive phases for prototype development, as depicted in Figure 2. They are Preliminary Analysis Modeling, Detailed Analysis Modeling and Modeling for Validation. This last phase uses two complementary sets of models, Prototyping Models and Testing Models.

ACCORD/UML starts the development process with a preliminary phase that consists of rewriting product requirements into an UML-based model. In this way, user requirements are better formalized through a vocabulary base with all the domain-specific concepts of the designed system. Regarding other object-oriented approaches such as [Dou99, JCJ92, RBP91], our contribution focuses on several key points: description of a dictionary consisting of a four-category structure; rules for naming model elements; modeling of real-time constraints; and modeling rules for classifying actor roles.

**Compiling the dictionary.** The dictionary is a compilation of all key concepts extracted from the initial requirement document. The dictionary has two main purposes: (i) familiarize the engineer in charge of

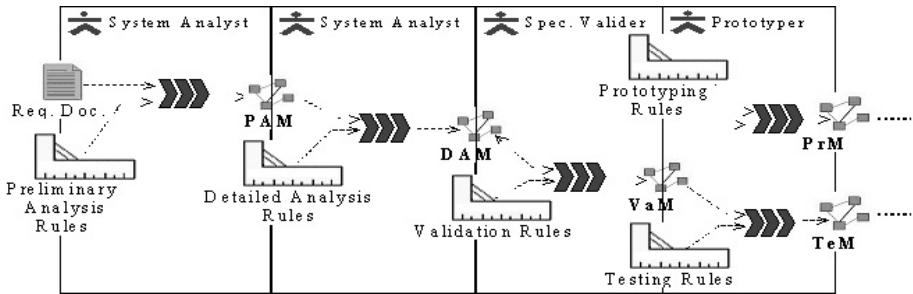


Figure 4.2. Overview of the development phases of ACCORD/UML methodology.

building the UML model with key concepts that are relevant to the current application (this avoids possible misunderstanding of concepts due to, for example, past experience in other domains); (ii) introduce preliminary modeling concepts such as classes, attributes, operations, relations.

All the terms used to describe the system are grouped under four headings: *Name* (defines concepts that correspond to classes or actors), *Qualifier* (can be associated with attribute or relationship concepts), *Verb* (often translated into operations and imposes relationships between two concepts) and *Real Time constraints*.

Table 4.1. Extract from the dictionary built for the seat controller application.

<i>Name</i>	<i>Qualifier</i>	<i>Verb</i>	<i>RTconstraint</i>
SeatControlSystem		Acitivated Deactivated	
Motor Groups		Move Stop	within 20ms
Heating System		Heating Stage 1 Heating Stage 2	last 30m last 5m
Hall Sensor		Indicate ticks	
Seat Switches		Pressed Released	
EEPROM	Store Position		

In addition to helping familiarize designers with a specific domain, a dictionary may serve as basic input for a tool dedicated to tracing user requirements through successive modeling stages.

**Describing use case model.** UseCase diagrams model behaviour that accomplishes some system functionality without detailing the internal structures of the entities. “Use case model building” refers to three steps: (i) identify environment; (ii) identify services; (iii) and identify relationships. There is no fixed order for these steps. In fact, a use case diagram can also be constructed by iterating them.

**Identifying environment.** Because systems never operate in isolation, the first job of the designer is to identify all external entities that interact with the system. We call this set of external entities the “environment”. In UML, such model elements are known as Actors. Table 2 shows the actors corresponding to our example in bold text.

Table 4.2. External entities interacting with the system

---

... <b>Hall sensor</b> which indicates the movement ...
... seat is adjusted by means of special <b>seat switches</b> ...
... only one <b>motor group</b> may be activated at a time ...
... the seat position is stored in <b>EEPROM</b> ...
...store position, <b>memory button</b> is pressed...and within 2 sec., desired <b>storage button</b> ...
... connected to car <b>environment</b> ...retrieve information on power, road speed or door sensor
... heating system is activated by one of 2 <b>heating buttons</b> ...

---

In ACCORD/UML, models are used to generate code in the last stage of development. To facilitate traceability analysis of specification and implementation models (required after renaming model elements to compile the product code), our approach calls for concept names specified in models to be “compilable” as soon as possible in the modeling process. For this reason, the following modeling rule is proposed: all names are concatenated by removing spaces and diacritical and a punctuation mark; then the first letter of each word is changed to upper case. This rule is not mandatory and may be adapted to the used language.

Table 4.3. Seat Controller actors with their matching system names

<i>Actor type</i>	<i>System name</i>
Hall sensor	HallSensor
Seat switches	SeatSwitches
Car environment	CarEnvironment

**Identifying services.** Services are the main functionalities a system provides to users. They may be identified in the dictionary through elements in the verb column. In our case study, there are six important functions depicted in Table 4. Major seat control system services: *adjusting*, *storing* seat position, *restoring* a position, affording *courtesy adjusting*, *heating* and *calibrating*.

Table 4.4. Major seat control system services

- 
- ... seat is **adjusted** by means of special *seat switches* ...
  - ... the *seat position* is **stored** in *EEPROM* ...
  - ... to **restore** position, press the desired storage button ...
  - ... **courtesy adjustment** is activated when door is opened and ignition key ...
  - ... **heating** the *seat* is activated by one of 2 *heating buttons* ...
  - ... the *seat* is **calibrated** so that all adjustment axes ...
- 

**Identifying relationships.** The last step in construction of a use case diagram is to identify relationships between use cases and actors. Such relationships specify communications taking place between actors and system for performing related use cases. The definitive use case diagram of the seat control system is described in Figure 3.

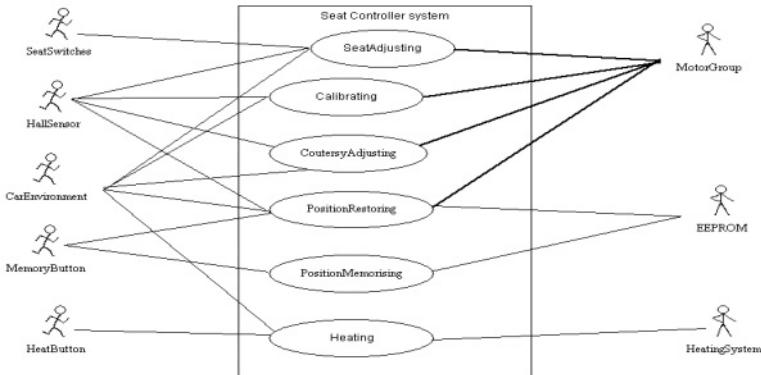


Figure 4.3. Complete use case diagram for seat controller system.

In this final use case diagram, communication points between the system and its environment have been identified, but the types, meanings and protocols of such interactions are not yet specified. Before describing how to do that, the following section focuses on real-time feature modeling.

**Modeling real-time constraints.** Real time constraints are used to specify real-time quality of services. In ACCORD/UML, they can appear at any analysis level. At the preliminary level, after extracting these constraints to the dictionary, designers can integrate them into high-level sequence diagrams and even the precedence diagram (see section 3). In ACCORD/UML-based applications, real-time constraints are modeled using the stereotype <<RTF>> (Real Time Feature) and its associated tagged values (Table 5).

For a periodic request, designers can consider the reference time *Tref* to be increased with an interval *Period* after every execution and *PeriodNumber* then indicates a fixed number of periods. An activity will not be reactivated for the next period if it has already been activated *PeriodNumber* of times. Specification of real-time constraints in UML and, more generally, in UML-based methods for real-time, is still informal and such constraints cannot, therefore, be formally connected with the other parts of the model. Taking them into account requires developers to use low-level instructions within the implementation code. This means that changing the way they are implemented also results in modifications to the modeling phase. In ACCORD/UML, however, real time constraints can be declared early in the model in the above format, which is independent to implementation. A change in implementation does not then require redesign or reanalysis of the complete application; instead, implementation choices are simply “tuned” as required in the last phase of real-time application design. Moreover, real-time constraints are implemented automatically with the ACCORD/UML kernel in the prototyping phase (section 5). A few examples of *RTF* stereotype usage are given later in this paper.

Table 4.5. RTF stereotype and its associated tagged values specification (ACCORD/UML profile extract)

Stereotype	BaseClass	Tags	Tag Type	Tag Multipl.
<<RTF>>	Action	Tref	RTtimeValue	[0..1]
	Transition	Deadline	RTtimeValue	[0..1]
	Message	Priority	RTtimeValue	[0..1]
		ReadyTime	RTtimeValue	[0..1]
		Period	RTtimeValue	[0..1]
		PeriodNumber	integer	[0..1]
		continueFunc.	booleanExpr.	[0..1]

**Describing high-level scenarios and precedence constraints.** A use case describes a behavior without detailing the internal structure of the involved entities. To achieve this, use cases may be associated with interactions in the context of a realization relationship, and interactions may be modelled in UML using either collaborations or sequence diagrams. Since our applications are part of the real-time domain, it is advocated to use sequence diagrams to model all possible interactions of a use case. Figure 4 gives an example of a single scenario matching the *StartAdjustment* use case. If an interaction is complicated due to

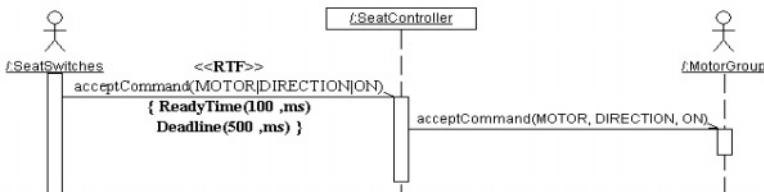


Figure 4.4. “start Adjustment”.

repetitions of certain sequences and (or) there are alternative sequences at a given moment, it is preferable to break down the interaction into simpler “sub-sequences”. To specify the order of these sequences for an entire use case, precedence relations are specified through UML activity diagrams as illustrated in the figure 5 for *SeatAdjusting*.

The *Reactivated* sequence is started when the driver inserts a key and switches on the ignition, provided no unusual events such as *overSpeed*, *underVoltage*, *overVoltage*, etc. take place. *StartAdjusting* is executed when he pushes one or more of the adjustment buttons. Then for the sequence *AcceptTicks-DetectBlocking*, *SeatControlSystem* executes the “detect blocking” periodic task and periodically receives ticks sent from *MotorGroups*. This use case can be subsequently terminated using one of three possible Stop sequence scenarios: driver releases adjustment button; motor reaches five ticks before end position; seat is blocked and system receives no more ticks for 250 ms. The real-time constraint “`<<RTF>> {Tref = (now, sec), Deadline = (250, ms)}`” in the precedence diagram in figure 5 specifies that the scenario “*Stopping*” has to be achieved within 250 ms of *Tref = now* (“now” being the key word of the ACCORD/UML profile depicting current time).

#### 4. Detailed Analysis Model (DAM)

In the previous phase, the focus was on describing the system external view through use cases that model interactions with environment. The

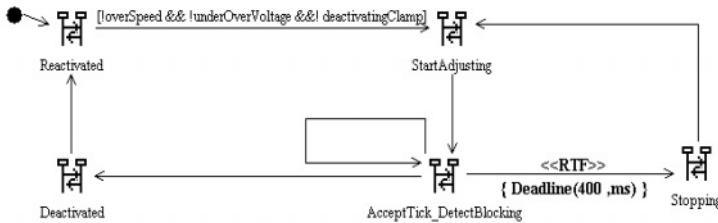


Figure 4.5. Precedence diagram of SeatAdjustment use case.

PAM thus serves as a black-box model of the system. Now, the DAM stage calls for exploring the inside of this box and DAM corresponding model describes PAM content in minute detail. From the black box point of view, the process progresses reached a “white box” phase. In this approach, the DAM can be organized around three partial, but complementary and consistent “sub-models”:

- *Structural model*: specifies the structure of the application in the form of class diagrams;
- *Detailed interaction model*: describes all possible scenarios in detail (contains a set of sequence diagrams);
- *Behavioral model*: specifies logic behaviors of the application via state-transition diagrams and algorithmic behaviors via activity diagrams.

This phase is automatically initiated by classes and actors in PAM. Designers subsequently complete it by adding all necessary information to the three types of these sub-models to describe application requirements in detail.

**Building a structural basis.** To facilitate the development of an application in term of reusable components, generic software architecture is constructed that separates the system core from its relationships with its environment. As shown in figure 6, this architecture relies on the following packages, which focus on describing:

- Provided interfaces - describes interaction points where the environment “stimulates” the system;
- Required interfaces - describes points at which the system interacts with its environment and models the environmental interfaces required for the system to operate;

- Core elements - contains all internal classes of the system application;
- Signals - groups all signal definitions in separate packages, to avoid confusing different concepts. *InternalSignals* contains signal elements falling within the scope of the Core package, whereas *ExternalSignals* is for those likely to cross system boundaries in both directions (i.e. either from external systems to current system or vice versa);
- Domain types - collects all specific domain types required to describe system requirements in an abstract way, without using implementation types such as integer, real, etc. For example, the SeatController system will have a TimeValue type domain for obtaining or storing the system time value. Designers can decide on the precise format of this type in the prototyping phase, not the requirement analysis phase.

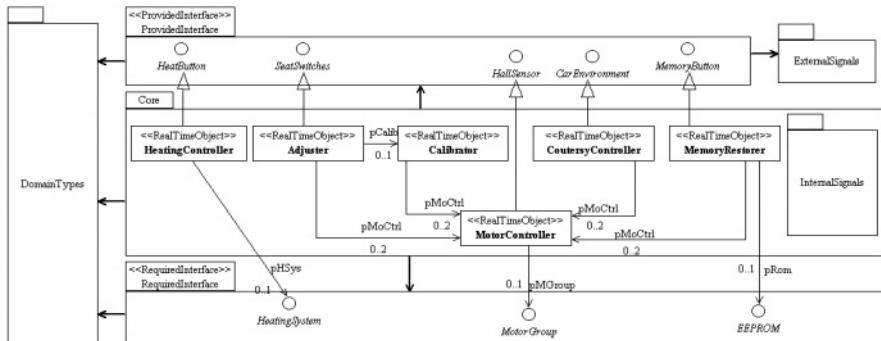


Figure 4.6. High-Level Architecture of an ACCORD/UML-based application.

In the SeatController system, there are six identified classes: *HeatingController*, which controls heating facilities; *Adjuster* for activation-deactivation of global seat adjustment (see protocol state machine and trigger state machine of *Adjuster* figure 9 and figure 10). The latter is linked with two *MotorController* instances and one *Calibrator* instance. Each instance of *MotorController* controls seat adjustment along one axis. The system requires two instances of this class because two adjustments are possible at a time. *MotorController* is linked with the required interface *MotorGroup* for sending commands to seat motors.

**Declaring active resources.** In ACCORD/UML, concurrency is handled by specification of Real Time Object [24]. A real time object in figure 7 consists of a mailbox saving incoming requests and a concurrency and state controller for managing message constraints according to its current state and the concurrent execution constraints attached to the messages. Moreover, real-time objects own a pool of threads that support the execution of the activities triggered by incoming messages. In contrast, passive objects have no resources for executing methods and thus use the resources of the caller objects. They do, however, also have concurrency and a state controller to manage parallel access conflicts. In a seat control system, all class instances become real-time objects

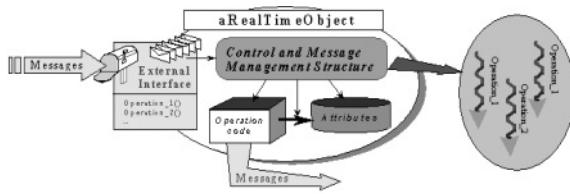


Figure 4.7. Real-Time Object concepts.

because they must be able to trigger autonomous treatments, receive signals, receive asynchronous calls, acquire data from the system environment and monitor periodic or sporadic behaviors of this environment.

**Specifying communications.** In UML, messages may be sent in either of two forms - as operation calls or signals. In ACCORD/UML, communication modes are specified and defined by the following:

- Operation-call based communication - which requires a structural link from sender to receiver. Operation-call mode is implicitly determined by its signature. By default, an operation having no return value or output parameter is invoked in *asynchronous communication* mode. Asynchronous communication means that senders immediately continue their execution without waiting for sent message treatment. An operation with output parameters is invoked in *delayed synchronous mode*. This mode permits the sender to continue execution, but if the return value is needed for some calculation, it must wait until the called operation is completed. This synchronization mechanism is achieved by a specific system object: a reply box shared by a sender and a receiver. An operation having a return value is invoked in *synchronous mode*. These three modes facilitate and maximize parallelism in real time system modeling [24].

- Signal-based communication - In ACCORD/UML, this mode may be used for broadcast-type asynchronous communication. In such case, the sender does not know the signal targets and receivers do not know the sender. This solution affords a more modular design, but also decreases structuring of the application while decoupling entities. All signals in the system must be declared in the application's signal package. As depicted in figure 8, interface class *CarEnvironment* can send four signals: *normalSpeedS*, *overSpeedS*, *setClamp15CS*, *setClamp15S*, real-time class *HeatController* is sensitive to two signals *setClamp15CS* and *setClamp15S* and similarly *Adjuster* is sensitive to its four arriving signals. Real-time constraints can be attached directly to any signal. For example, a real-time constraint is attached to arriving signal *overSpeed* of *Adjuster*,  $\langle\langle RTF \rangle\rangle\{Tref = now, Deadline = (300, ms)\}$  to specify that the reaction of this signal must take place within 300ms.

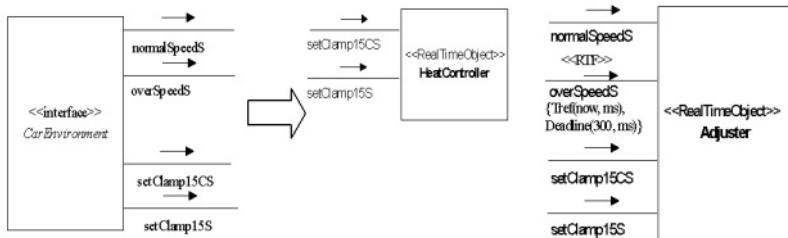


Figure 4.8. Extract of signal sending and receiving specification.

**Describing behavior view.** Since our approach targets the real-time domain, the behavior model used must be deterministic. The ACCORD/UML profile contains several rules for specifying the open points in UML semantics. These serve as “formalizing” rules for statemachine semantics to ensure suitable determinism and executability features. To describe the behavioral aspect of an application, ACCORD/UML has also introduced an approach based on two complementary views: protocol view and reactive view [18], [25].

*Executable semantics for a state machine* In UML, events are dispatched and processed one at a time but the dequeuing order is not defined, so users have possibility of modeling different dequeuing schemes as needed. In ACCORD/UML, a message always possesses a real-time feature (see section 3) that is specified either explicitly or implicitly by constraint propagation [24]. Dequeuing policy is specified by using the mailbox semantics. This point is described in detail in section 4.

Priority-based or deadline-based dequeuing methods are possible, but cannot be mixed in a same application. The event having the strongest processing constraints (i.e., highest priority or shortest deadline) is selected from the mailbox and then processed. ACCORD/UML methodology has specified a set of additional rules using OCL in order to formalize clear and non-ambiguous semantics for UML state machines. Due to space limitations, this rule set cannot be described in detail in this paper. Readers can refer to the ACCORD/UML profile for more information on these rules.

*Protocol view:* This view defines the context and the order in which call operations may be called. To enhance the expressive power of models in the ACCORD/UML approach, designers can use the three following mechanisms for treating unexpected received messages: *immediate rejection* with generation of an exception (using keyword rejectedEvent), *immediate discard* (with keyword ignoredEvent) and *placing on standby* (default operating mode).

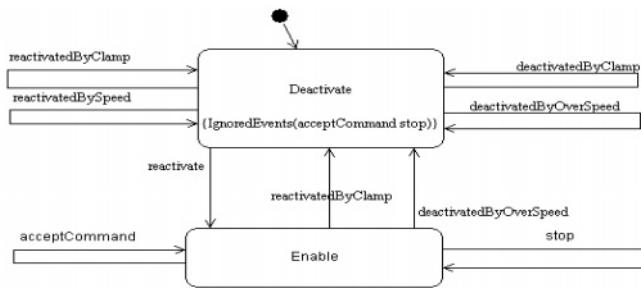


Figure 4.9. Extract of protocol state machine diagram for Adjuster class.

*Reactive view:* A reactive view focuses on the reactivity of a class. A transition is fired when one of three following events occurs: (i) change event when a boolean expression becomes true; (ii) time event on expiry of a specific deadline; (iii) and signal event to which class was declared sensitive in the structural model.

However, designers must comply with a specific rule here: the operation triggered on receiving an event must already exist in a similar transition in the protocol state machine. This means that the protocol view has to own a transition with same source and target state and this transition is triggered by a call-event linked to the operation specified in the action part of the reactive transition. As shown in figure 10, on the triggered *overSpeedS/deactivatedByOverSpeed* transition from Enable to Deactivate state, there is a real-time constraint

$\langle\langle RTF \rangle\rangle \{Tref = now, Deadline = (300, ms)\}$  for specifying the

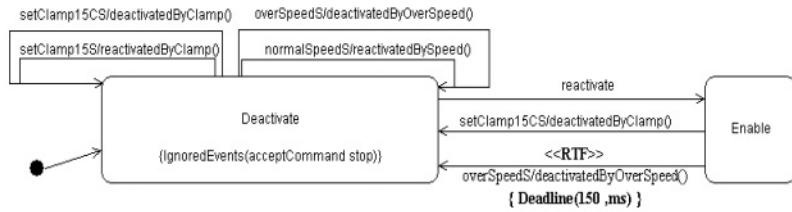


Figure 4.10. Extract of Trigger state machine diagram for Adjuster class.

deadline of the response time of operation deactivatedbyOverSpeed when an instance of object Adjuster receives the signal overSpeedS. A full parameter example of real-time constraints can be found in figure 12.

**Building a scenario basis.** In the system as it now stands, there are many objects that interact with each other to perform services. At this stage, designers therefore need to describe in detail the messages exchanged by them for every initial scenario specified in the PAM. This point can be illustrated by detailing the “startAdjusting” scenario in Figure 4 as in figure 11. Figure 12 depicts a detailed sequence diagram for

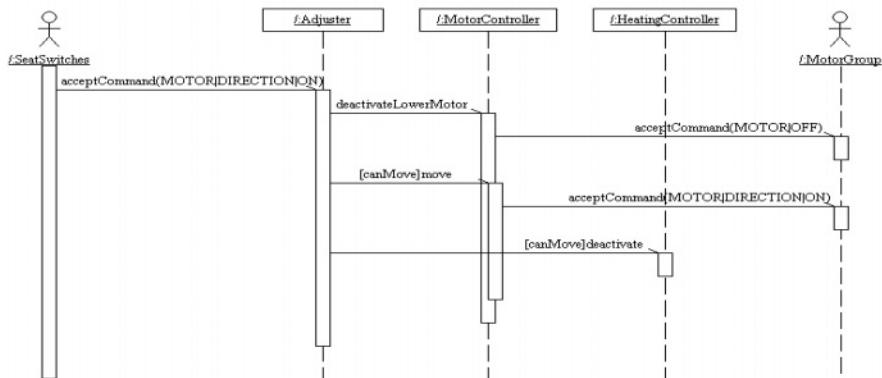


Figure 4.11. Detailed sequence diagram for StartAdjustment.

modeling real-time constraints. In the *AcceptTicks-DetectBlocking* sequence, the following real-time constraint is modeled:  $\langle\langle RTF \rangle\rangle \{ Tref = (now, ms), Deadline = (20, ms), Period = (200 ms), ContinueFunction = pContinueF \}$  for operation *detectBlocking*. Under this real-time feature, the operation is assigned current time as reference time, then is immediately started and must be finished within the 200 ms deadline. It is also periodically executed (i.e. once every 200 ms). The last tagged

value attached to *RTF* is a boolean expression that is evaluated at every end of period. As long as it is evaluated to true the method continues to be executed, but should it become false, the periodic treatment is stopped. Here, *pContinueF* returns false when a seat-blocking event is detected or when the system receives enough ticks before the “axis end” position.

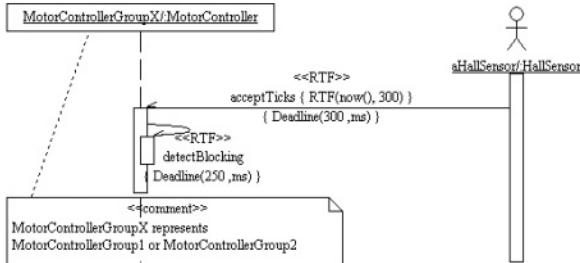


Figure 4.12. AcceptTicks\_DetectBlocking sequence diagram with real-time constraint

**Summary of DAM.** In response to the question “what internal functions will the system afford?”, the ACCORD/UML approach calls for organizing global model construction around three partial, but complementary and consistent “sub-models”: a structural model, a behavior model and a detailed interaction model. These three sub-models, when taken together, constitute the complete application specification.

## 5. Validation by Prototyping (PrM) and Testing (TeM)

ACCORD/UML offers two options for validating models: either construction of an application prototype through assisted synthesis of specific design patterns and automatic code; or use of a formal tool called AGATHA for automatic test case generation. The latter entails first mapping ACCORD/UML models using AGATHA input language based on the paradigm of concurrent automata communication by “rendezvous”. Due to space limitations, the next sections of this paper can only outline this particular aspect of ACCORD/UML methodology. For more information on both subjects, see [27], [28].

**Building a Prototype Model.** The ACCORD/UML method is supported by Objecteering (trademark of SoftTeam). The available configuration involves a set of modules allowing generation of the real-time behavior of an object from its state machine specification into C++ code.

Moreover, a specialized C++ generator has been developed to support the concepts of the extended active object. Finally, with respect to the underlying real time operating system (RTOS), two layers are available between the application and the RTOS, namely ACCORD Kernel and ACCORD/VM (Virtual Machine). The first of these implements mechanisms supporting active object semantics and specifically permits scheduling of application tasks in compliance with EDF (Earliest Deadline First) policy. The second ensures application independence from the underlying RTOS, which is a POSIX OS such as Solaris or VxWorks5.2. Within these two layers, designers must add real-time constraints for call operations and signals that still have not these constraints to obtain a complete real-time application. All system behaviors are examined and verified when executing this compiled application. If an error (a task that missed its deadline, incorrect behavior, etc.) is detected, the relevant real-time constraints or the system model must be modified to ensure suitable operation of the final application.

**Building a Test Model.** In addition to the previous prototyping facility, there are several other means for system validation, among them theorem proving and model checking [21]. Such techniques have proven successful for validation of system critical parts. However, they still have two major drawbacks: combinatorial explosion due to variable domains for model checking; and need for highly skilled developers with suitable grasp of fundamentals inherent in formal theorem proving methods.

Use of the AGATHA tool set [28] allows automatic generation of a Test Model (TeM) for the current application. AGATHA uses symbolic simulation that accommodates variable domains, since computing all behaviors is not equivalent to testing every possible input value. It is therefore helpful to exhaustively compute system symbolic dynamic behaviors as specified in state machines for classes in the model. AGATHA then automatically generates tests by way of constraint solving. With this symbolic execution tool, it is possible to verify whether sequence diagrams specified in PAM and DAM phases are compatible with the set of sequence diagrams computed by AGATHA. If incorrect sequence diagrams are identified, the system model must be modified to improve its operation.

## 6. Conclusion and ongoing research projects

Reusability and evolutivity are the main requirements for modeling and development techniques. UML and UML-based methods partially meet these requirements. Nevertheless, they still lack sufficient semantics for a complete description of system real-time behaviors. In addition, the

development process does not yet have a uniform, continuous supporting methodology for quick, easy design and adaptation of application products. By integrating the active object paradigm into an object-oriented method, ACCORD/UML affords simple, complete and uniform modeling of requirements. ACCORD/UML extends UML to real-time by enriching this paradigm for real-time concerns. Use of the same object paradigm throughout the development cycle reinforces seamless transition from one modeling stage to another. This approach enables designers to truly benefit from object-oriented experience acquired in other domains.

Because the constraints related to multitasking and real-time behavior of the application can be expressed early in the model itself, then implemented automatically, designers do not require in-depth knowledge of the fine synchronization mechanisms used in multitask programming. The ACCORD/UML approach likewise permits postponement of all parallelism, real-time and implementation choices until very late in the design process. Changes in parallelism granularity thus do not lead to redesign or reanalysis of the entire application, but just require the tuning of implementation choices of the last real-time design of the application.

In conclusion, ACCORD/UML provides a suitable environment for design of real-time embedded systems. It has been evaluated by users in the automotive and telecom industries, as part of the European Union's AIT/WOODDES [19], [29], [30] project. ACCORD/UML allows designers to master the increasing complexity of application products and to adapt such products easily and quickly to market developments. PhD research underway on ACCORD/UML methodology should, in the near future, enhance the approach described here, by (i) enabling integration into the ACCORD/UML platform of a module to assist engineers in analyzing schedulability [31]; (ii) affording further improvements to embedded code generation facilities; (iii) allowing support for the family of products paradigm in ACCORD/UML; (iv) providing a clear method and related tools for sensor modeling and automatic implementation of sensor drivers . Ongoing ACCORD/UML research is also directly relevant to the MDA (Model Driven Architecture) concept developed by OMG [29], [32]. A new large-scale joint research program (CARROLL), involving CEA, INRIA and Thales ([www.carroll-research.org](http://www.carroll-research.org)) is now focusing on the definition and development of MDA components for use in design, implementation, validation, operation and maintenance of DRES.

## References

- [1] Unified Modeling Language Specification, version 1.4. OMG. 2001.
- [2] S. Gerard, F. Terrier and B. Selic. UML for Real-Time, in UML for Real: Design of Embedded Real-Time Systems, Kluwer Academic Publishers: Boston. p. 369, 2003.
- [3] UML Profile for Schedulability, Performance and Time. OMG. 2003.
- [4] RFP on “UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms”. OMG. 2001.
- [5] A. Moore and N. Cooling. Real-time Perspective- Overview (v 1.0): ARTISAN Software. 1998.
- [6] M. J. McLaughlin and A Moore. Real-time Extensions to UML. Dr. Dobb's, 1998.
- [7] A. Moore. Why task aren't objects and how to integrate them in your design. Embedded Systems, pp. p18-30, 2000.
- [8] R. Arthaud. OMT-RT: Extensions of OMT for better Describing Dynamic Behavior. In TOOLS Europe'95. Versailles, France. 1995.
- [9] P. Leblanc. Object-Oriented and Real-Time Techniques: Combined Use of OMT, SDL and MSC. Current Issues in Electronics Modeling Series, 1996.
- [10] R. Z. ITU-T. Specification and Description Language. In ITU-T. Geneva. 1996.
- [11] V. Perrier. Combining Object-Oriented and Real-Time Programming from an OMT and SDL Design. Verilog France, Toulouse, 1997.
- [12] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. ObjecTime Limited, 1998.
- [13] C++ Language Add-in, C++ Language Guide. ObjecTime Limited. 2000.
- [14] B. P. Douglass. Doing Hard Time : Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. 1999.
- [15] Real-Time UML: Developing Efficient Objects for Embedded Systems. B. P. Douglass. 1998.
- [16] D. Harel. Statecharts: A Visual Formalism for Complex System. Science of Computer Programming, vol 8, pp. p231-274, 1987.
- [17] F. Terrier, D. Bras and F. Fouquier. Proposition of ACCORD project about Real-time objects and deadline scheduling. 1996.
- [18] S. Gard. Executable modeling for automotive embedded systems development, in GLSP. Evry: Paris, 2000.

- [19] CEA, I-Logix, Uppsala, et al. Methodology for developing real-time embedded system, document M2.1 of the IST AIT-WOODDES, 2002.
- [20] Daimler Chrysler. Seat adjustment specification <http://www.automotive-uml.de>. 2000.
- [21] E. M. Clarke, O. Grumberg and D. A. Peled. Model Checking. The MIT press, 1999.
- [22] I. Jacobson, M. Christerson, P. Johnson, et al. Object-Oriented Software Engineering: A Use Case Driven Approach. 1992.
- [23] J. Rumbaugh, M. Blaha, W. Premerlani, et al. Object-Oriented Modelling and Design. Prentice Hall, 1991.
- [24] F. Terrier, G. Fouquier, D. Bras, et al. A Real Time Object Model. TOOLS Europe'96, Paris, France, 1996.
- [25] C. Mraidha, S. Gerard, F. Terrier, et al. A Two-Aspect Approach for a Clearer Behavior Model. In ISORC'2003. Hakodate, Hokkaido, Japan: IEEE. 2003.
- [26] S. Gerard, E. Pelachi, P. Petterson, et al. Methodology for developing real time embedded systems. CEE: Paris. p. 158, 2002.
- [27] A. Lapitre. Procédure de réduction pour les systèmes à base d'automates communicants : formalisation et mise en œuvre, in UFR Scientifique d'Orsay. Université de Paris XI: Evry, 2002.
- [28] D. Lugato, C. Bigot and Y. Valot. Validation and automatic test generation on UML models: the AGATHA approach. FMICS Malaga, Spain, 2002.
- [29] S. Gerard. The ACCORD/UML profile. CEA-LIST, 2002.
- [30] I-Logix, Thales, Tri-PACIFIC Software, et al. UML profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms - Initial proposition. OMG, 2002.
- [31] T. H. Phan, S. Gerard, D. Lugato, et al. Scheduling Validation for UML-modeled real-time systems. In WiP of ERCT. Portugal. 2003.
- [32] J. Bezivin and S. Gerard. A Preliminary Identification of MDA Components. OOPSLA 2002 Workshop: Generative Techniques in the context of Model Driven Architecture, 2002.

## Chapter 5

# UML-BASED SPECIFICATIONS OF AN EMBEDDED SYSTEM ORIENTED TO HW/SW PARTITIONING

## *A Case Study*

Mauro Prevostini, Francesco Balzarini, Atanas Nikolov Kostadinov, Srinivas Mankani, Aris Martinola, Antonio Minosi

*Advanced Learning and Research Institute (ALaRI),  
University of Lugano, Switzerland, www.alari.ch  
mauro.prevostini@unisi.ch*

**Abstract** The Unified Modelling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artefacts of software systems, as well as for modelling business and other non-software systems. The UML represents a collection of best engineering practices that succeeded in modelling large and complex systems; it is interesting to envision its extension for specification and modelling of hardware-software systems as well, starting with the first design phases, i.e. prior to hardware-software partitioning. This paper analyses the development of a solution able to define the hardware/software partitioning of an embedded system starting from its UML system specifications. The case study chosen is a Wireless Meter Reader (WMR) dedicated to the measurement of energy consumption. The designers evaluated the hardware/software partitioning solution in terms of cost, performance, size and consumption.

**Keywords:** System-level Design, UML, HW/SW Codesign

## 1. Introduction

As the complexity of systems increases, so does the importance of good specification and modelling techniques. There are many additional factors for a project's success, but having a rigorous modelling language

standard is certainly an essential factor (see, e.g., [5], [9]). In recent years, the Unified Modelling Language, UML, has been introduced and is now widely used, basically for requirements specification in the design of complex software systems. UML does not guarantee project success but it does improve many things. For example, it significantly lowers the perpetual cost of training and retooling when changing between projects or organizations. It provides the opportunity for new integration between tools, processes, and domains. Most importantly, it enables developers to focus on delivering business value and provides them a paradigm to accomplish this. In the present paper we describe use of UML for specification and modelling of a hardware-software embedded system, carried out at the Advanced Learning and Research Institute (ALaRI) of the University of Lugano. In this paper we will propose a solution that elaborates UML specifications and use them in order to support designers' decision making process for hardware and software partitioning. This paper will present the motivation of using UML for Hardware and Software partitioning in Section 2, while Section 3 discusses the problem description of the case study in terms of objectives, and operational scenario. Section 4 describes the WMR System-level specifications using UML showing use cases diagram, sequence diagrams and object model diagram developed for this case study. Section 5 explains the proposed UML-based HW/SW partitioning approach of the system-level specification. In Section 6 you will find our concluding remarks.

## **2. Why Hardware and Software Co-design starting from UML**

The pervasiveness of embedded systems, in each operation we use to do everyday in our life, has an important impact in projects development lifecycles. Despite the 21st century crisis, the embedded system market is still rapidly growing following ICT market [6], determining the importance of time-to-market products delivery. Attention to details, good specification and modelling techniques and hardware/software co-design methodologies are instrumental to grant product quality under short delivery time. Current approaches used by the industry are still unsatisfactory. Embedded software is still written by hand and most often developed from scratch each time a system is extended or upgraded. This behaviour requires tremendous efforts in terms of development resources determining significant cost increment in both development and research centres. Industry pressure to reduce time-to-market points to the need of looking for ways to change the current situation [7]. In order to solve this problem, during past years, the software community

converged on a sets of notations for specifying, visualizing, constructing, and documenting artefacts of software systems: the Unified Modelling Language (UML).

Use of UML is now spreading also for embedded systems combining hardware and software components, which must be accurately specified. During the last two years there has been a lot of interest in the idea to specify SoC's using UML. Recently a new community has been created under the name of "SoC Design with UML" [10]. What is still missing from our point of view is the automation of co-design methodologies starting from UML system specifications in the embedded systems world. A related article to the present work is [11] which describes a first step towards HW/SW partitioning with UML.

Mixing HW and SW development steps is a central point of the process between project specification and solution deployment: dedicated HW can be introduced to overcome impossible or too expensive SW solutions, which are usually cheaper. Two ways can be followed to link UML to the partitioning problem:

- UML to SystemC
- UML to direct partitioning

UML let us generate specification for a lot of different problems, so it should be directly considered the specification language for an embedded system. Problems arise when embedded systems designers ask to get something practically usable from UML and they don't get anything because there is no hardware synthesis available. The advantage of using UML is that it makes the whole designed system highly modular, so that we could structure the whole embedded system as the interconnection of a number of different blocks consisting of Hardware and Software components. It was decided to use UML because it meets the following high-level requirements specification:

- It is technology independent;
- It allows the top-down approach;
- It precedes HW/SW partitioning and in fact it is designed to support the partitioning phase;
- It allows specification of both functional and non-functional requirements and constraints;

The case study and the approach chosen for our project are described in the following sections.

### 3. Case Study: Problem description

WMR is a reading system to help utility providers to get consumption data regarding gas, electricity and water via wireless technology.

#### 3.1 Objective

The objective of this project was to design a device able to perform real-time determination of energy consumption (where, when and to what energy is consumed), using wireless technology in a low-power and low-cost environment. As described in Figure 5.1, the meter reader is the embedded system we are analysing, whereas meters are traditional devices dedicated to energy consumption measurement. Usually meters are located in home basements where GSM signal is not available. The Meter Reader is the low-power device which aims to solve this problem by reading, on a regular basis, data measured by meters and sending them (e.g. once a day) to a Data Collector (DC) through a wireless Bluetooth connection. The DC, located where the GSM signal is available, collects data and store them in the local memory. Then the DC sends these data to the utility provider using SMS. As well as that the utility (or service) provider should be able to read at any time data stored in DC and, if necessary, determine the real-time meter data.

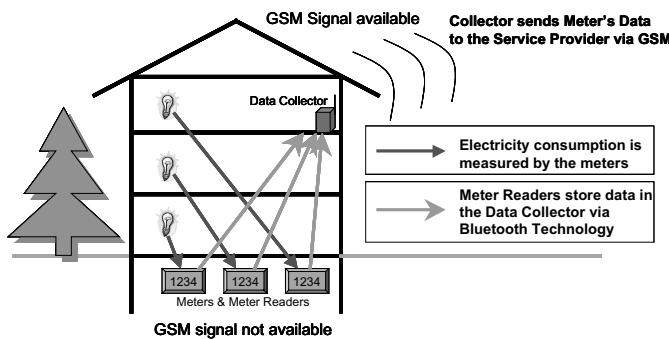


Figure 5.1. Problem description. This model is valid also for water and gas meters.

#### 3.2 The operational scenario

**The Wireless Meter Reader features.** Figure 5.2 provides a summary of the overall system, in which the WMR is inserted, and its functionality. In order to better describe the WMR features, as shown in Figure 5.2, we divided the system in two sub-systems corresponding to the End-Consumer and the Service Provider side.

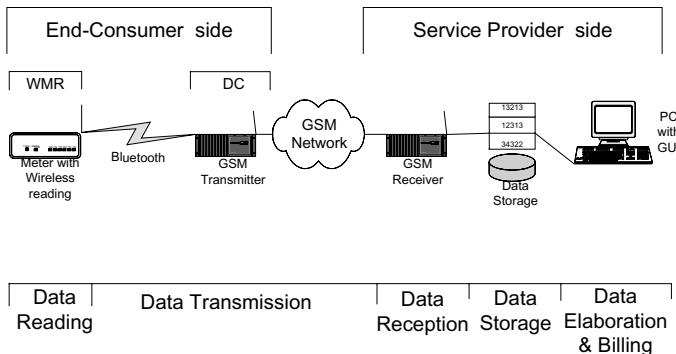


Figure 5.2. High-level system description of the WMR System.

**Features at the End-Consumer side.** The data reading operation can be performed in two different ways depending on the meter type: electronic or mechanical (traditional) meter. In the first case where, at the End-Consumer side, an electronic meter is installed, it is sufficient to introduce suitable sensors in order to make the system able to read meter values. In the second case (mechanical/traditional meter) the meter reading and radiation can be done using an Optical Detection Method: the image of the meter digits is transferred by simple mirror and lens technique to a CMOS image sensor. Both solutions are possible. Data will be transmitted using standard data formats provided by AMRA [3], which is an international and non-profit association addressing problems of standardization, justification and deployment practices in the application and advancement of enhanced customer-service and resource-management technologies. The data transmission architecture does not directly connect the meter to the GSM network, for a number of different reasons. Usually, meters are located in basements, so that the GSM signal reception is very weak or even not present at all. To avoid this problem we considered a transmission architecture using a wireless network that brings data to the GSM transceiver located a few meters away from the basement, where GSM signal is available. In order to grant a low-power and low-cost solution, we chose to build a Bluetooth network. Bluetooth is one of the technologies that can be used for transmitting data from a given meter location point to the GSM transmitter. Bluetooth devices are high-quality, low-cost and low-power devices: usually the chip-set cost is lower than 10 USD and each unit is self-powered. The transmission of the measured data to the Service Provider side can be performed using SMS. The fragmented nature of the telemetry and telecom markets has given rise to a wide variety of

technology alternatives, from low-power radio to landlines. A recurring theme in these markets, however, is that where GSM has been available, it has been widely and successfully employed. Measured data will be transmitted from the Bluetooth unit to the GSM base station.

**Features at the Service-Provider side.** Data reception is done through the GSM network. One or many stations located at the provider's side can be used for receiving data and for load-monitoring. Data received will be stored in a database, developed ad-hoc for each Service Provider or integrated with existing systems. The data management at the utility side is performed by a GUI developed ad-hoc that displays collected data. In case the utility has already a system in place, this functionality is granted by the integration with the existing system.

### 3.3 The project constraints

The constraints of the project are related to security, low-power consumption, meter reader identification and status, data reading frequency and system re-configurability:

**Security.** The WMR must grant security requirements at the data integrity level, theft monitoring and eventually at encryption level. Basically the WMR should be able to ensure data transmission to the Data Collector without packet-loss: packet retransmission will be ensured if corrupted data will be received. In case of theft monitoring, WMR should be able to send a message to the Service provider if it detects not authorized manipulations.

**Low-power consumption.** As the WMR system is intended to be a device placed at remote locations, it should be a low-power and low-cost device in order to stay alive as long as possible. We performed a low-power analysis and we decide that WMR would be in a power-safe mode most of the time, but in this case there is a risk of missing messages. In order to minimize this risk, the Bluetooth transceiver of the data concentrator transmits a prelude frame containing the address of the enquired meter before the transmission can be established. The transmission time will be longer for the Bluetooth transceiver of data concentrator (DC), but this device is not under strict power consumption constraints like the radio transmitter on the meter.

**Meter reader identification and status.** To each Meter Reader a unique address (e.g. IPv6) must be assigned in order to grant the right identification and location in case they are moved from one location to another. Determining the MR status is also important in case of damage theft actions. A theft monitoring functionality has been foreseen.

**The reading frequency.** There is only one time base for the whole system (standard time DCF 77). A quarter of an hour is the smallest measuring period. Longer measuring times are multiple of a quarter of an hour.

**System re-configurability.** The meter reader device is designed in order to grant a good flexibility level in order to support the utility requirements. It often happens that utility providers change with time, the billing policy, and this usually has a relevant impact on the measurement system configuration. In fact billing models change even between utilities so that the measurement system should be flexible enough. In our WMR we have foreseen a bi-directional Bluetooth connection allowing changes in the Meter Reader software configuration. Configuration changes allowed are first of all related to the meter reading frequency. The utility should be able to decide which is the time window needed for their billing model. In this way, each utility is able to customize the meter reading frequency in order to avoid problems when global pricing politics change the way to bill energy consumption. Low-power constraints, on one side, and the foreseeable use of semi-permanent memories (e.g., Flash Memories) requiring more power for writing than for reading, on the other side, outline a reconfiguration practice that should be limited both in frequency and in relevance (e.g., concerning updates of tables, or at most substitution of small library segments).

## 4. WMR System-level Specification with UML

UML is a modeling language rather than a methodology. It is largely process-independent; in fact it is not tied to any particular development life cycle. However, to get the most benefits from the UML, one should consider a process that is: **Use Case driven, Architecture-centric** and **Iterative and incremental**. This use case driven, architecture centric and iterative/incremental process can be broken into the Rational Unified Process (RUP) phases [8]: inception, elaboration, construction and transition which can be applied very well in the development process of this project. The UML tool used was Rhapsody V4.1 from I-Logix.

### 4.1 Use Case diagrams

We worked out a big use case diagram that specifies main activities involved in the AMR project. Actors in our diagrams are the Service Provider (SP) central station, a Data Collector (DC) that is installed in buildings, a WMR (target of the partitioning solution) and a sensor with its related meter (counter). SP keeps a centralized database to collect all information that DC inside houses should send once per month. DC must daily collect data in houses. Data should already be encrypted by

WMR with a digital signature. WMR should keep a little database and collect meter's data each quarter of an hour. Theft monitoring issues are centralized in WMR by exploiting additional sensors to detect manipulation of the principal one. Correct timing is important: the system should work for years interacting with the SP only by Short Message System (SMS), and WMR should read data starting at an exact time to avoid phase shifts with the billing policy. As example we introduced this issue by considering the possibility for the DC to adjust its time with the global timing signals coming from satellites and managing to update the WMR timer counter from the DC side to override clock skews in the WMR itself that is battery powered. Reconfiguration may be required

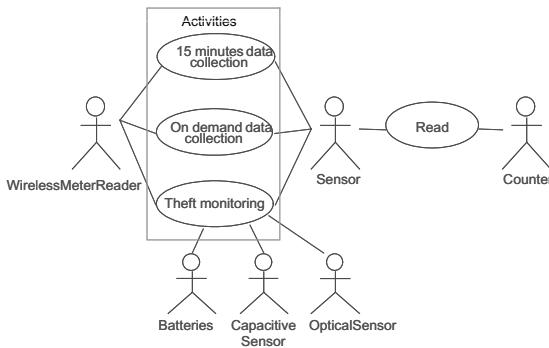


Figure 5.3. Use Case Diagram of MR Activities.

by SP to have different periods of data readings and to adapt the system to the chosen billing policy of each installation. It is also provided a way to let SP ask an immediate data collection and a way to let theft monitoring messages to be immediately sent from DC to SP.

**Wireless meter reader and sensors activities.** As described in Figure 5.3, we specified capacitive and optical sensors as generic theft monitoring sensors to detect wires or packaging manipulation, moreover battery voltage is tested to be able to send information to the SP before batteries exhaust. So WMR is the most delicate part of the project because of high number of pieces that have to be produced and because of the amount of intelligence involved also for managing a plethora of counters and related sensors that have various interfaces.

## 4.2 Sequence Diagrams

**Sequence diagram for data collection issues.** Figure 5.4 describes the sequence of signals for four possible data collection scenarios. First of all

we described *Monthly data collection*: data is sent every month from Data Collector to Service provider. To be sure that data is received we used *AcknowledgeNewData* signal. The second sequence described is *Daily data collection*. In this case *WirelessMeterReader* sends information to *DataCollector*. For security issues we proposed data to be encrypted before sending. The third sequence refers to *15 minutes data collection*. In case of traditional (mechanical) meters, data are read every 15 minutes through the optical sensor. In case of digital meters we skip using optical sensor. Service Providers are also able to ask for Real-time data: the sequence is shown in the *OnDemandDataCollection* scenario.

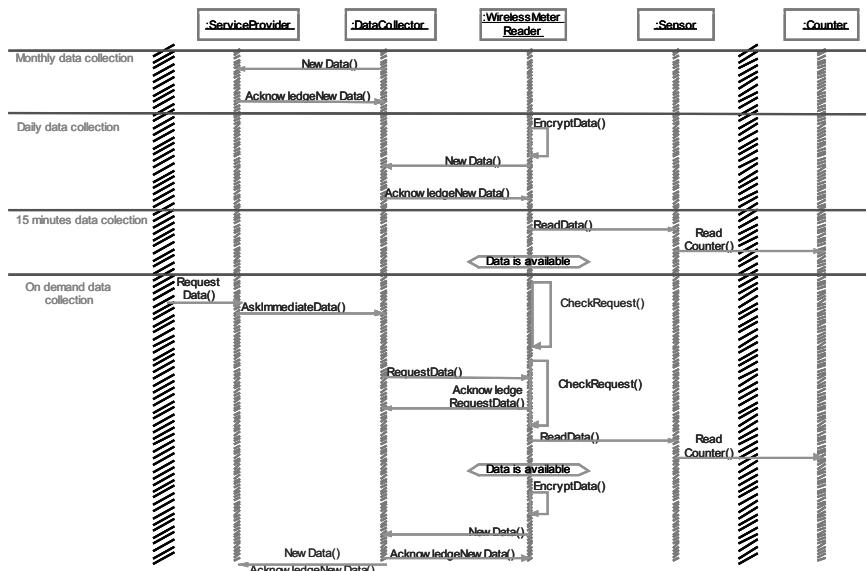


Figure 5.4. System Sequence Diagram.

**Sequence diagram for system re-configurability.** As billing policies changes, we specified a scenario where it is possible to reconfigure the WMR remotely. This functionality gives WMR enough flexibility to allow Service Providers choosing between a few prepared configurations. Thus utility companies are allowed to change the reading time interval in order to exploit different billing policies (See Figure 5.5).

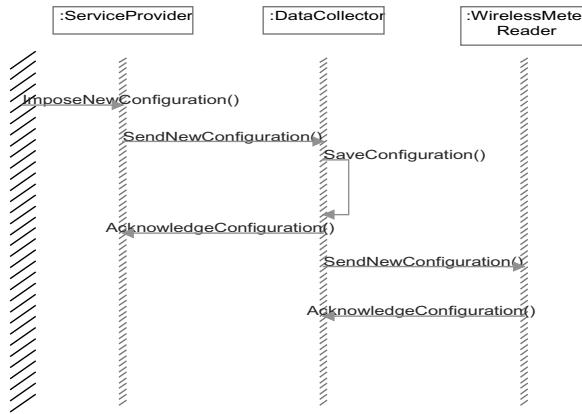


Figure 5.5. Sequence Diagram for Re-configurability.

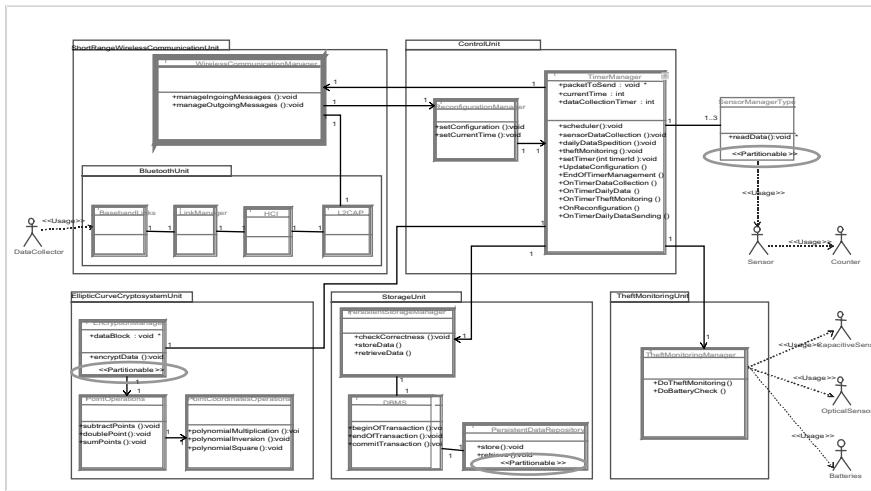


Figure 5.6. Object Model Diagram. The ‘Partitionable’ stereotypes are marked with a green ellipse.

## 4.3 Object Model Diagram

As already stated, we put our efforts in the description of the WMR unit because it is the most demanding in terms of run-time efforts constrained to low-power problems.

## 5. UML-based Hardware and Software Partitioning Approach

As the reader could have noticed, until now we didn't introduce partitioning concepts or strange constraints in the usage of UML to specify a design: in fact it *is* what we wish. Our approach is a transparent introduction of partitioning concepts just by attributing the stereotype *Partitionable* (see Figure 5.6) where the designer wants to check the system feasibility or cost or whatever without sticking the design phase with some prohibitions. The integration between UML output and a component repository is left apart and the partitioning tool must operate apart, too. The steps that we foreseen can be exploited to make a range of different evaluations over the design: parameters are not predefined and the *partitioning engine* can be exploited to make other kinds of analysis; for instance: if the designer doesn't introduce constraints on packages, object types or objects, the result is just a one by one allocation of components accordingly to what has been chosen and result is just the cost of the chosen configuration. Figure 5.7 describes a general

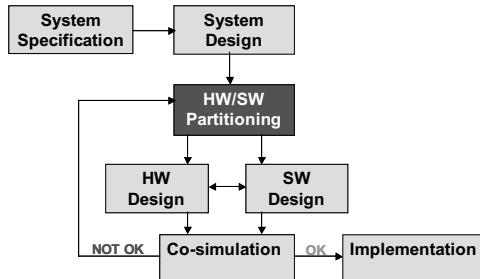


Figure 5.7. Partitioning is part of HW/SW Codesign.

HW and SW Co-design methodology where Partitioning is an important step to be performed. In order to generate input parameters for Partitioning tasks, in the following sections (from 5.1 to 5.6), we propose the steps to be followed, starting from UML specification through system design, for a whole partitioning tool usage methodology.

### 5.1 STEP 1: Assign the "Partitionable" stereotype to desired objects, object types and packages

Stereotypes are a simple extension mechanism of UML model: they can be user-defined and can be seen as additional information for classes, usually they are used to attribute some implementation hints (*Task*,

*Semaphore, Web page...):* in our case we use it to tag UML graphic elements that are going to be considered by the partitioning algorithm. This approach doesn't block the UML development process and makes our tool usage orthogonal to the design process. Figure 5.6 describes the object model diagram where some classes are tagged as "Partitionable" like:

- *SensorManagerType*;
- *EncryptionManager* belonging to the *EllipticCurveCryptosystemUnit Package*;
- *PersistentDataRepository* belonging to the *StorageUnit Package*.

## 5.2 STEP 2: Assign parameter's constraints

Constraints are specified in text format inside the description field of UML elements. We plan to move all of them in a separated file to let the UML specification be even less influenced by partitioning algorithm. The simplified syntax can be seen here with a mixture of "Backus Naur Form" [4] and regular expressions, it doesn't really correspond to what is implemented to make it more readable (Bold are keywords, italic are trivial terminals):

- Constraint:  $I([packageId:][objectId:(methodId:)][parameterId:][\mathbf{max} - \mathbf{min}]Value(Unit);)+]$
- Value:  $[integerNumber - floatingPointNumber]$
- Unit:  $[\mathbf{n}\text{-}\mathbf{p}\text{-}\mathbf{u}\text{-}\mathbf{m}\text{-}\mathbf{c}\text{-}\mathbf{K}\text{-}\mathbf{M}\text{-}\mathbf{G}] [\mathbf{s}\text{-}\mathbf{m}\text{-}\mathbf{W}\text{-}\mathbf{Hz}\text{-}\mathbf{USD}]$

Constraints will be considered later in the design space exploration to produce suitable rules for the Integer Linear Programming (ILP) problem.

## 5.3 STEP 3: Parse the UML saved files

This is the UML tool dependent part, until eXtensible Markup Language (XML) Metadata Interchange (XMI) is universally adopted we need to parse saved files and rebuild a tree of meta-classes corresponding to the UML model, then locate elements with the *Partitionable* stereotype and extract sub-trees with some other details as the number of instances for certain objects that can both improve the accuracy of results and set extra constraints. Parser outputs are both a file with all constraints associated with component names and a little amount of simple rules that express implicit constraints coming from UML specification.

## 5.4 STEP 4: Assign parameters to components from a repository or attribute parameters by hand

Selected components must be mapped to elements inside a repository that collects the know-how of the company that adopts our approach. Parameters can also be assigned to blocks without any previous study: in this case we planned to keep just the extreme values (all HW implementation and all SW implementation) and, in order to detect possible working points, impose both a granularity and a simple interpolation rule to get possible intermediate implementations that have to be considered. Elements taken from the repository have to be prepared by another team of workers dedicated to create the low-level know-how of the company and must be tuneable in terms of cost parameters used by the company.

## 5.5 STEP 5: Decide cost function to give weights to parameters

Cost function is an important detail and it is responsibility of the designer to choose the most suitable weighting factors for the parameters that he needs to estimate. In a company the good choice of cost estimation depends on designers experience and know-how. A good approach to compute the global development effort, measured in person/month, for realizing a given system is COCOMO2 [1].

## 5.6 STEP 6: Run the partitioning tool

This is naturally the last step needed to get results. Partitioning tool will use specified constraints to detect parameters that have to be retrieved from our repository to prepare the ILP problem that will be passed to one of the freely available ILP problem solvers.

# 6. Concluding Remarks

The characteristics of the proposed methodology are basically the following:

- *Not intrusive*: stereotypes can be easily used without impacting the UML system specifications; in text format it's possible to specify constraints in an easy way.
- *Suitable to approach complex designs*: in case of complex object model diagrams it considerably simplifies the collections of those parameters needed for the HW/SW partitioning tool.

- *Can be effective from early design stage:* in embedded system design, co-design is a very important step. So that the opportunity to specify, in early design stage, parameters for the HW/SW partitioning step, it dramatically decreases risks of introducing bugs during the design phase.

The usage of this methodology before HW/SW partitioning implies a good UML knowledge as well as a good experience in determining parameters values for cost functions. Reliability in using this methodology grows with company's know-how. Low-level block's characterization is already available by other tools, like POLIS [2].

## References

- [1] Cocomo 2.0. 1997. *Model Definition manual, ver 1.2.*
- [2] Polis, a design environment for control-dominated embedded systems. 1999. *User's Manual, version 0.4.*
- [3] Automatic Meter Reading Association (AMRA), <http://www.amra-intl.org>.
- [4] What is BNF notation?, <http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html>.
- [5] D.Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8):231–274, 1987.
- [6] EITO 2003. Press releases, <http://www.eito.org/press-releases.html>.
- [7] J.L.Diaz-Herrera, J.Chadha, and N.Pittsley. Aspect-oriented uml modeling for developing embedded systems product lines. *School of Computing and Software Engineering, Southern Polytechnic State University, Marietta, GA*.
- [8] P.Kroll, P.Kruchten, and G.Booch. *The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process*. Addison Wesley Professional, 1st edition, 2003.
- [9] S.Narayan, F.Vahid, and D.d.Gajski. System Specification with the SpecCharts language. *IEEE Design and Test of Computers*, pages 6–12, 1992.
- [10] UML for SoC Design, <http://www.c-lab.de/usoc>.
- [11] W.Fornaciari, P.Micheli, F.Salice, and L.Zampella. A first step towards hw/sw partitioning of uml specifications. *Politecnico di Milano, CEFRIEL, 2003*.

II

## C-BASED SYSTEM DESIGN

*This page intentionally left blank*

Embedded systems are becoming an integral part of daily life in developed countries. As the functionality and performance of the single Silicon die increases while its cost decreases, a growing number of products containing such devices are produced. Examples of them range from mobile phones to ABS brakes in cars. System specification constitutes an essential step in any system design methodology. It is the step in which the designer envisages the whole set of intended characteristics of the system to be implemented. The lack of a unifying system specification language has been identified as one of the main obstacles bedeviling SoC designers. The problem crops up because traditional languages like VHDL for the HW part and C for the embedded software are no longer valid to capture the entire functionality of the system, now composed of functions to be implemented either in HW or in SW. Moreover, the complexity of each part demands new specification paradigms supporting efficient code generation, code profiling, real-time analysis, verification, test strategy, HW/SW partitioning, SW extraction and compilation, HW extraction and synthesis, interface synthesis, co-simulation, etc.

Since its introduction in 1999, SystemC is gaining a wide acceptance by the users' community as a system-level specification language supporting the basic features required for such an application. Today, with the availability of release 2.0, SystemC is on the verge of entering the industrial world. Nevertheless, in order to become a mature, system-level design technology, there are still several aspects, which have to be completely fixed and understood. SystemC lacks features to support embedded software modeling. This means that systems with hard real-time constraints requiring an RTOS (Real- Time Operating System) based on a pre-emptive priority-based kernel cannot be modeled naturally. Such RTOS provide a very useful abstraction interface between applications with hard real-time requirements and the target system architecture. As a consequence, availability of RTOS models is becoming strategic inside HW/SW co-design environments.

The complexity of modern designs requires the engineer to approach testing problems from the early stages of the design flow. Testing activities performed at high abstraction levels require techniques and methodologies mixing concepts derived from both the verification and the testing fields. SoCs are characterized by their heterogeneous nature. Heterogeneity affects both the architecture and the functionality. Depending on the characteristics, each function may be better specified and designed using a specific Model of Computation (MoC). Therefore, in order to

allow the designer to use the most appropriate specification method for each domain, the design language used should support several models of computation. In some cases, heterogeneity demands the use of several languages at different stages of the design flow. ECL is a system design language suitable for heterogeneous, reactive system specification and design.

The C/C++-Based System Design (CSD) Workshop of FDL'03 once again provided a perfect place for a fruitful discussion on all the problems related with system-level specification and design. The CSD Section of the book provides a selection of the most relevant works presented during the Forum. They cover all the previous concerns: RTOS modeling, high-level test pattern generation, practical use of SystemC, models of computation and system design from ECL.

So, in Chapter 6, a SystemC platform for architectural exploration, named SPACE (SystemC Partitioning of Architectures for Co-design of Embedded systems) is presented. It is a top-down approach that first lets designers specify their application in SystemC at a high abstraction level through a set of connected modules, and then simulate the whole system. This untimed simulation allows fast functional verification. Then, the application is partitioned in two parts: software and hardware modules. Each partition can be connected to our platform that includes a commercial RTOS executed by an ARM ISS scheduled by the SystemC simulator. A separate simulator for hardware and software partitions enables a timed co-simulation. One of our major contributions is that module movements from hardware to software (and vice-versa) can be done easily, with the only requirement of recompilation.

In Chapter 7, a high-level Test Pattern Generation (TPG) methodology for SystemC designs called Laerte++ is described. All necessary features of such a tool (e.g., fault model definition, hierarchical analysis, coverage measurements, etc.) are implemented by exploiting native SystemC characteristics founded on Object-Oriented principles. Laerte++ allows the set-up and running of the TPG procedure by adding very few C++ code lines to any SystemC DUT description.

The paper in Chapter 8 describes the modeling and refinement process of an industrial application from the automotive domain starting from a high-level C description down to a cycle accurate SystemC model for hardware synthesis. This real design example allows the study of the design flow, starting from an algorithmic C description (which is often used for a specification of the system behavior) down to synthesizable SystemC, which is the starting point for further design activities, such as synthesis to hardware and implementation on an appropriate target platform. The case study includes a comparison of the C and SystemC

implementations, as well as an analysis and discussion of the refinement process and the hardware synthesis task.

In Chapter 9, a general system-level specification methodology for SystemC is proposed. It follows an orthogonal separation between functionality and communication without limiting the specification capability of the designer. The specification methodology supports several computational models and even heterogeneous specification combining some of them. In the paper, three classic, fundamental MoCs are studied, specifically, Communicating Sequential processes (CSP), Kahn Process Networks (KPN) and Synchronous Reactive systems (SR). The work shows the flexibility of SystemC in supporting different MoCs when an appropriate specification methodology is used. It opens the way for a truly use of the language in the specification of heterogeneous systems combining several MoCs.

In Chapter 10, a HW /SW co-design framework for medium-size systems using ECL is proposed. The paper investigates the hardware synthesis problem for ECL provided that verification and software synthesis pose no special difficulties.

Eugenio Villar

*University of Cantabria, Spain*  
*villar@teisa.unican.es*

*This page intentionally left blank*

## Chapter 6

# SPACE: A HARDWARE/SOFTWARE SYSTEMC MODELING PLATFORM INCLUDING AN RTOS

Jerome Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois,  
El Mostapha Aboulhamid, Francois-Raymond Boyer

*Electrical Engineering Department, Ecole Polytechnique de Montreal, P.O. Box 6079, Succ.  
Centre-Ville, Montreal, Quebec, Canada, H3C 3A7*

**Abstract** This work attempts to enhance the support of embedded software modeling with SystemC 2.0. We propose a top-down approach that first lets designers specify their application in SystemC at a high abstraction level through a set of connected modules, and simulate the whole system. Then, the application is partitioned in two parts: software and hardware modules. Each partition can be connected to our platform that includes a commercial RTOS executed by an ARM ISS scheduled by the SystemC simulator. One of our major contributions is that we can easily move a module from hardware to software (and vice versa) to allow architectural exploration.

### 1. Introduction

During the last recent years, some efforts have been made to introduce object-oriented (OO) languages for system-level modeling, such as C++, to reduce the design and verification time by raising the abstraction level of system specifications. This gives better component reusability and is an acceptable proposition for managing the exponential growth of complexity of embedded systems.

It appears today that SystemC is the leader in system-level modeling with C++. The SystemC approach consists of a progressive refinement of specifications. The design cycle starts with an abstract high-level untimed or timed functional (UTF/TF) representation that is refined to a bus-cycle accurate and then an RTL (Register Transfer Level) hardware model. SystemC is not a design methodology but it does propose various

layers of abstraction that are useful for specification capture in the early stages of a design flow.

One of the problems encountered with SystemC 2.0 is the lack of features to support embedded software modeling. Of course, at a high level of abstraction (e.g. UFT/TF), SystemC allows the use of a common language for software and hardware specifications, and the simulation of the whole system. However, during the simulation, the scheduler, responsible for determining which thread will run next, manages identically both software and hardware threads. It means that systems with hard real-time constraints requiring an RTOS (Real-Time Operating System) based on a preemptive priority-based kernel cannot be modeled naturally. As a consequence, availability of RTOS models is becoming strategic inside H/S (hardware/software) co-design environments [2].

Thus, one of the main objectives of this paper is the development of a SystemC platform for architectural exploration, named SPACE (SystemC Partitioning of Architectures for Co-design of Embedded systems). SPACE enhances the support for embedded software modeling, by encapsulating RTOS functionalities into an API (Application Programmable Interface) allowing the use of a common language. In this work, MicroC/OS-II [10] has been selected as an example, but the mapping of the API can be easily modified to incorporate other RTOS.

As a consequence, the hardware and software modules can be specified with UFT/TF SystemC. Apart from the fact that software modules (threads) will require priority levels, all the services offered by the RTOS can be called through existing SystemC functions. Afterwards, software modules are compiled, and the obtained binary file (linked with the RTOS kernel) is executed on the processor. In SPACE, the processor is modeled by an ISS (Instruction Set Simulator) for which a SystemC wrapper has been added. The ISS and hardware modules are considered as part of the traditional SystemC simulation. The communication between hardware modules and software modules is done through an abstract communication protocol using a TLM (Transactional Level Model).

One of the main advantages of SPACE is its easiness to move modules from hardware to software (or vice versa) during the architectural exploration. Except for the thread priorities that could be modified, we only need to recompile and simulate.

The paper is organized as follows. Section 2 of this article discusses about the related work and underlines our objectives. In Section 3, we describe SPACE and its methodology. Section 4 describes in detail the embedded software environment, while Section 5 describes the communication channel interface. In Section 6, we show an application using

SPACE and we present simulation results. Finally, Section 7 presents the conclusion and future work needed to increase the functionality of SPACE.

## 2. Related Works and objectives

Currently in the majority of industrial projects, after the specification phase, what will be the software and hardware parts constituting the future SoC (System-on-Chip) is chosen following *ad hoc* methods, often based on the designer experiences. Then, the development of the hardware part and the software part of the SoC is performed in two disjoined design flows. This is problematic because errors appear very late in the design process and modifying hardware/software partitioning requires a huge amount of work. The reason is the lack of tools during the partitioning phase.

Several efforts were made to ease partitioning, by making possible the specification and simulation at system level, then refining it in an iterative way towards the final implementation. SpecC [4] and SystemC [12] support such an approach inside a unified specification language based on C and C++, respectively. However, for the simulation of software modules, the SystemC simulator does not offer all the necessary functionalities, such as preemption or scheduling by priority, generally present in any RTOS: a joint refinement of the software and hardware parts is thus a tedious task in SystemC 2.0. A possible area for consideration in extending the SystemC core language is to provide better software support [6]. Unfortunately, the specification for this future release is not yet available. Consequently with SystemC 2.0, there are tools able to synthesize hardware modules [9], but this is done at the expense of the software part. There are also methods to refine the software modules [6], based on POSIX threads, recognized as a soft real-time operating system (rather than hard real-time).

Several researches are thus concentrated on ways to simultaneously simulate software and hardware in a realistic manner with SystemC (or with C++) in order to perform the partitioning with a better understanding of the system. To consider the communication aspects between the software and hardware modules during simulation [15], we can intercalate an adapter. This adapter replaces the processor at high level, which will be selected in the final architecture. This makes possible the simulation of various architectures at high level. Nevertheless, since each processor has its own behavior, an adapter is required for each processor. Also, concerning the scheduling of the software modules, no OS (Operat-

ing System) implementation is supported: only foreground/background applications.

Another possibility is to use an adapter that provides RTOS properties [3]. This allows to schedule several software modules in a sequential way such as an RTOS would do it. It also allows fast simulation. However, this adapter generation is achieved on a golden abstract architecture resulting from the architecture exploration. Also, no RTOS model seems to be considered during the architecture exploration. Although this approach does not provide much help toward automating design, it reduces the design time considerably (by automatically generating devices drives, operating systems and APIs) and facilitates component reuse [3].

[8] proposes a SystemC code used for the system-level specification, and after H/S partitioning, for the embedded software generation. Then, modules of the software part are written in SystemC and can be simulated at high level. Also, a SystemC/RTOS interface that makes possible modules scheduling based on an RTOS is presented. But again, they do not propose any environment to facilitate the architectural exploration (further explanations will be provided in Section 4).

A possible refinement [3] is to simulate more accurately the software interaction with the hardware, using an ISS. The ISS is generally a hardware module simulated by SystemC that accepts a binary code obtained by the cross-compilation of the software modules. Several researches were already undertaken to integrate an ISS with SystemC [13, 1]. The results show that this integration is possible and that by using an ISS already written in C/C++ (like those provided by GNU), it is possible to quickly obtain a functional system. The resulting simulation is reliable and realistic because it depends on the actual architecture. Also, the use of an ISS at the transactional level simplifies the memory access method. This decreases largely the number of delta cycles necessary and accelerates significantly the simulation. On the other hand, these solutions focus more on the simulation aspects than on the partitioning methodology: the use of an ISS seems to take place after the partitioning phase. Furthermore, no proposal suggests the use of an RTOS making it possible to schedule several software modules on the ISS, so programs being executed remain of the foreground/background type, which is today a substantial limitation in SoC designs.

In summary, no method currently exists to easily explore and simulate various high-level hardware/software configurations, in order to obtain results leading to the optimum partition of a system. Two main conditions are required to reach this goal: the possibility of moving modules between the software part and the hardware part without changing modules' code, combined to a simulation of the whole system giving realistic

results to validate or invalidate a partition choice. The following section proposes a way to reach these two requirements: the use of an architecture simulated in SystemC integrating an ISS with an RTOS, and various mechanisms and interfaces of communication.

### 3. SPACE and its methodology

Our proposed methodology allows obtaining the ‘optimum’ partition of a system based on simulation results. First, the system is specified at the functional level (UTF/TF) in SystemC, but not partitioned. The construction of the modules representing the system must follow some rules, so that it will be eventually easy to move them from the hardware partition to the software partition (and vice versa). The methodology forces the module to use only thread constructions (i.e. no SC\_METHOD) and to have only one single advanced input/output port. The coding style of the modules is thus close to the behavioral style and is not significantly restrictive.

The first step is to simulate the system in a purely functional form with a SystemC transactional model named *UTF channel*, in order to check that it respects the functional aspect of the specification. Once the functionality validated, the next step is the partitioning stage. Modules tested previously are taken again, without modifications, and are placed in a simulation architecture, independently in the software or hardware part of the system.

As mentioned, we have chosen to use an ISS in our architecture to jointly simulate the software part with the hardware part. Similarly to [14], we use ISS models of the GDB debugger. As a first experiment, we have chosen the ARM processor. It has been encapsulated (wrapped) in a SystemC module with *clock*, *reset*, *IRQ* and *input/output* data ports. A SystemC *wait()* statement sensitive on the clock signal is added in the main loop of the ISS to synchronize SystemC with the ISS clock. The memory access functions of the ISS are redirected towards the *input/output* data port. Communication is achieved through this port using the *read()* and *write()* functions. These functions receive two arguments: address and data. This raises the memory bus abstraction level and preserves the memory mapped I/O functionality. As the memory (through its decoder) is the interface of this port, the reading and writing calls are ended up in only one delta cycle (except if we insert *wait()* to simulate a non-zero access time), which makes it possible to have a fast simulation.

Modules placed in the software part of the architecture are thus cross-compiled and the binary code is placed in a memory module to be executed by the ARM ISS. As we want to have several modules in the

software part, we also added the core of an RTOS, MicroC/OS-II, which will schedule the various modules. Moreover, as the modules are written in SystemC, an interface (API) has been created so that they can communicate with the RTOS. This RTOS and the interface are also cross-compiled with the modules since they belong to the code executed by the ISS. The details concerning the RTOS and the interface will be given in section 4. A number of modules are present in the architecture to ensure the correct operation of the ISS and the RTOS, such as the timer module and the interrupt controller module. Interrupts are necessary to ensure preemption between the software modules. Modules placed in the hardware part of the architecture are connected to the *TF Channel* through wrappers so that they can communicate between themselves. Software modules communicate through the API. Moreover, we inserted a decoder module between the ISS and the memory. This makes it possible to connect the ISS with the channel through the decoder and a wrapper. Therefore, as shown in Figure 6.1, software and hardware parts can communicate together.

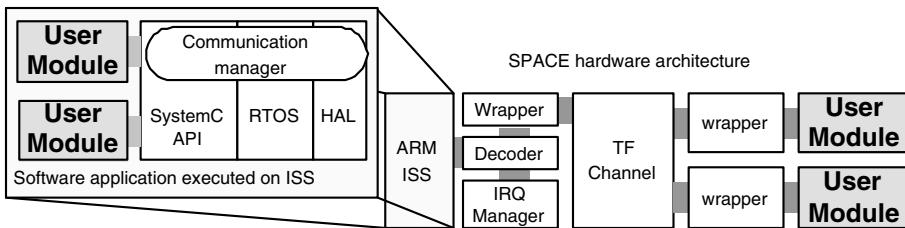


Figure 6.1. User view of the platform with the hardware and two software modules

## 4. Embedded Software environment

As mentioned previously, the SystemC 2.0 scheduler uses the same behavior for software simulation as for hardware simulation. Our goal is to provide the main services offered by the real-time kernel of an RTOS.

Rather than integrating an RTOS model directly in SystemC (similarly to [5] with SpecC), the proposed SystemC API allows an RTOS to schedule software SystemC modules. Also, it allows a simulation of the software part very close to reality and very early in the partitioning process. Another advantage is the possibility to change this operating system by another in order to satisfy the system's specifications.

The objective is to obtain a binary file that will be executed by the processor (initially an ISS) of the platform. Three parts constitute the binary code: the user application, the SystemC API and the RTOS.

## 4.1 SystemC API

First, the API uses the SystemC initialization mechanism to obtain the process list. Then, the initialization of the RTOS (MicroC/OS-II) is called. Each process generates the creation of a MicroC/OS-II task. Then, finally, the RTOS scheduler starts.

The SystemC API provides the mapping between the RTOS functions and the system functions proposed in SystemC 2.0. These functions manage processes, modules, interfaces, channels and communication ports. The work described in [8] uses a similar interface including an RTOS. The difference is that we focus on the partitioning mechanism in order to migrate modules between hardware and software seamlessly.

The SystemC API also provides a communication manager. It establishes connection between software modules and the platform (hardware part). It answers task requests to communicate with other software or hardware modules. The construction of the modules representing the system follows a ‘standard’, so that it will be eventually easy to move them from a hardware partition to a software partition (and vice versa). Any communication to a module will have to use a specific channel with specific methods: *read()* and *write()*. The aim of the software communication manager is to model communication between connected modules. This manager provides the same communication model than the hardware channel.

Through the manager, a module can access peripherals directly (memory, timer, etc.) or can send a message to another software module or to a hardware module. In the last case, the manager can communicate with a specific hardware module on the platform and then transfer the message to it. Similarly, modules can receive messages from software modules, or hardware modules. To receive a message from a hardware module, an interrupt is triggered and then a routine is executed to receive the message. The message is then sent to the communication manager so that the software module can get it.

Finally, note that SystemC provides several data types, in addition to those of C++. These data types are mostly adapted for hardware specification. In order to support hardware/software partitions, the SystemC modules will have to be functional in the hardware part as well as in the software part of the platform. The data types specific to hardware are implemented in the software SystemC API to allow compatibility during the simulation in the software part. It implies that all SystemC keywords are overloaded and all hardware specific data types, redefined.

## 4.2 The RTOS

The first RTOS to be integrated in our platform is MicroC/OS-II [10]. It offers all the advantages of a real-time operating system: a pre-emptive kernel, a priority based task scheduler and an interrupt system. MicroC/OS-II has been selected for its low complexity, the availability of its source code and because it has successfully been ported to a vast range of processors.

## 5. Hardware support

To ensure that communications will be preserved whether we decide that certain modules will be hardware and others software, each module has a unique identifying ID number, given by the system designer. Communications work as on a network and data are encapsulated in a packet with a structured header that contains sender's ID, target's ID and the size of the message, so that they can be routed correctly. If software and hardware modules want to communicate together, a special device called the *ISS Adapter* is used. Other useful devices such as RAM, interrupt manager and timer are also provided with the platform. There is a difference between what is named *Modules* and *Devices*. Modules can initiate transactions towards other modules or devices, created by designer. Whereas devices are slave blocks, they only give response to module requests, provided with the platform.

### 5.1 Abstraction level

Since an ISS is used in our architecture, a cycle accurate hardware channel to perform timed simulations is an interesting feature. Even though our timed channel (*TF Channel*) is synchronous and may contain an arbiter, it remains at a functional level, because the simulation must be fast and must abstract RTL details. We also support a faster way of communication named *UTF Channel* that performs a complete simulation of an application, before the partitioning phase, where all modules and devices are linked together.

Before describing in details our communication mechanisms, it is significant to discuss their level of abstraction. Four abstraction layers, L3 to L0, are recognized [7]. Our *UTF Channel* corresponds to level L-3 (Message layer), while our *TF Channel* matches best the level L-1 (Transfer layer).

At the Transaction layer (L-2), communications can be timed but are not cycle-accurate. If we wanted to perform a simulation at this level, we could add *wait(simulation\_delay)* calls into users' modules connected

to our UTF Channel. As L-2 is not yet implemented in SPACE, for the moment we omit this abstraction level in our methodology. However, an application developed and tested at a high level of abstraction (L-3) can be integrated easily on the platform (L-1), without refining by the intermediate level L-2. Compared to L-1, the main advantage of L-2 is a simulation time speedup.

## 5.2 UTF Channel

The UTF Channel is useful at the highest level in the design process. At this level, it is not significant to consider if the modules described in SystemC will be implemented in software or in hardware. The goal of the UTF Channel is to allow a quick verification of an application. To reach this goal, communication between several modules must take a reasonable amount of time. The way to achieve this is to keep the communication model as simple as possible, and to focus on untimed message passing. The modules are interconnected and simulated without suffer from Slow ISS. This feature enables us to validate the system without the whole platform, i.e. without a bus protocol, a microprocessor and a real-time operating system.

## 5.3 TF Channel

Following the design of an application that is carried out on high level, with the UTF Channel, we propose to replace the communication mechanism with another one: the TF Channel. The main refinement between the UTF Channel and the TF Channel is that the data transfers consume clock cycles.

Hardware/software partitioning can be obtained following the analysis made with simulations on the platform, at the TF level. To evaluate multiple suitable partitions, one of the main selected measures is the number of needed clock cycles for a complete execution of an application. For an application already conceived and verified on the UTF Channel, the user will be able to test various software and hardware configurations. For each configuration, the user starts the simulation and stops it at one precise moment, then notes the number of clock cycles used at this time (for instance by using the function *sc\_simulation\_time()*). The hardware/software configuration that is estimated as being the best is, in general, the one that will satisfy timing constraints, while minimizing hardware. The software modules consume clock cycles while being executed, because the processor takes a precise number of execution cycles for each assembly instruction. The hardware modules are considered as being powerful calculating units and consume clock cycles when they

communicate or when the designer uses explicitly a *wait()* statement. Finally, area estimation could be performed using commercial tools [9].

In SPACE different communication models could be evaluated. For the moment only the bus and the crossbar models have been implemented. To emulate a bus behavior, processes wait a certain amount of time that is proportional to the message size to be transmitted. Moreover, the bus must be used by only one process at a time, because it is a shared resource. This characteristic tries to emulate a single bus for all the modules. To simulate the serialization of TF Channel requests and answers, an arbiter is used. In this way, there cannot be more than one process transferring data on the TF Channel at the same time. The protocol delays have been modeled, but kept very simple: only the transfer time and arbitration time is calculated. Arbitration takes 1 clock cycle and transfer takes 1 cycle per 32 bits chunk.

Figure 6.2 gives an overview of the platform's architecture. We can see that there are adapters between user modules and the TF Channel. These components will enable us to have an independent communication interface for modules thus making it possible to use a different bus protocol if desired. Adapters are also essentials in our proposed architecture because they are used as message buffers. They can retain *read()* calls so that only *write()* calls will travel on the TF Channel. This enhances performance since half of the TF Channel traffic is eradicated.

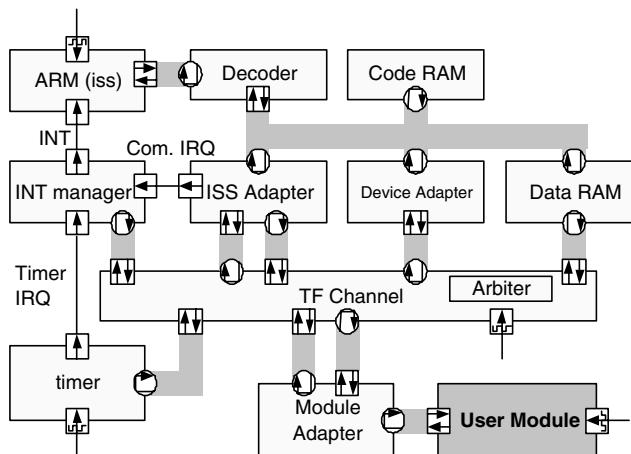


Figure 6.2. General hardware architecture

## 6. An example and its simulation results

An example is elaborated to illustrate the benefit of SPACE and its methodology. We could imagine such an application for audio or video data processing. Figure 6.3 is a functional diagram of the example and illustrates the five constituting modules and their communication dependencies. Integer data is first produced by the Producer module, filtered by the Filter module and stored in a buffer (reserved memory range) by the Mux module. Periodically, the Controller module wakes up and tells the Mux to store data somewhere else in the memory. Then the Controller asks the Analyzer module to use previously stored data to produce a result from a simple calculation. The result is needed to adjust filter coefficients. After that, the Controller waits until its next execution period. The example has been written following our methodology. First,

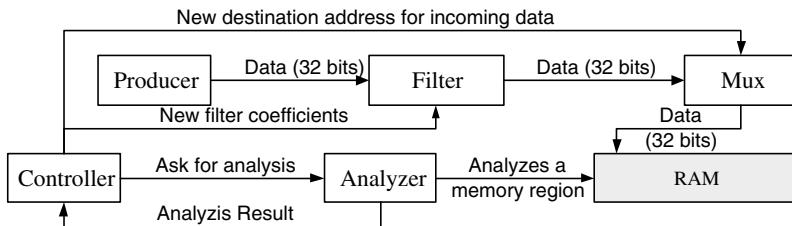


Figure 6.3. Example with the SPACE platform

we programmed the application in SystemC and bound modules to our high-level untimed fast simulating communication channel (i.e. the UTF Channel). Next, with negligible effort, we bound the same modules to the SPACE platform. In that way, we obtained a 100% hardware version of the application. The next version obtained is the opposite partition, i.e. 100% software. Finally, we tested a partitioned approach: the Controller and Analyzer modules constitutes the software part while the Producer, Filter and Mux modules constitutes the hardware part.

Simulation results are presented in Table 6.1. For every version, 10240 integers (32 bits) are produced by the Producer module. Results show that simulation is very fast using the UTF Channel. This allowed us to debug and verify the functionality of the whole system. Despite the fact that this version is untimed, we use a SystemC `wait()` statement in the Producer thread to model a periodic data generator. Also, the same strategy is used in the Controller to ensure a periodic execution. This is why the simulated time (in clock cycles) is non-zero using the UTF Channel.

With the next version, 100% hardware at the TF level, communication between modules consumes clock cycles and we see that simulation time is affected. The simulation is a bit slower because we introduced a global clock that synchronizes every thread. On the other hand, this version provided more accurate simulated time results.

The 100% software version is also presented. This version has been produced by compiling modules with the MicroC/OS-II library for the ARM ISS and our SystemC API. As expected, the application takes many times the needed cycles (compared to the hardware version) to perform the same task. Results from this version could be taken as an upper bound for simulated time.

Finally, the last version shows that both hardware and software threads can execute in parallel to provide interesting simulation results. By moving the Controller and the Analyzer modules to the software side and keeping the Producer, the Filter and the Mux modules in the hardware part, similar performances (less than 20% more cycles than the hardware version) can be obtained. This also results in a much more reasonable simulated time increase.

*Table 6.1.* Simulation results for the example (Pentium III 600 MHz, 128 MB)

<i>Example versions</i>	<i>Simulated cycles</i>	<i>Simulation time (sec)</i>
UTF Channel	2.3245 E06	1
100 % hardware (TF)	4.6912 E06	42
100 % software (TF)	2019.6 E06	6120
Partitioned system (TF)	5.6262 E06	66

Many other partitions could be tested and one that satisfies time constraint specifications could be chosen as the final system.

## 7. Conclusion and future works

In this work, we have shown how our architecture with an ISS and an RTOS creates the link between two abstraction levels and provides an easy and practical way to partition an application, based on simulation results. It allows high-level simulation of hardware/software applications coded in SystemC by respecting the behavior of the two parts while being sufficiently precise to validate a partition choice and by allowing the permutation of the application modules.

Simulation results could give more information to the system designer if we integrate a software profiler and hardware surface and power consumption estimators. The use of an ISS is essential to support full RTOS functionalities and provide precise results. However, similarly to [3], we are currently looking for a software layer that could provide high-level software emulation and faster simulations. This layer could be incorporated in our methodology as an earlier step in the application refinement process. The platform architecture is for the moment very simple but we intend to try other RTOS, processors, and channel protocol models [14, 7] and extend the architecture to support multiple processors. Finally, it could be interesting to incorporate existing high-level models of other communication protocols. It is possible for example to create a model that is functionally equivalent to OCP (Open Core Protocol) [11], without however modeling all the structures normally needed by the protocol [14]. That would make it possible to have a model which is fast to simulate and which in spite of this reflects reality in terms of clock cycles (cycle true). Having several channel models to choose from could also allow exploration of the system interconnect.

## References

- [1] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. Legacy SystemC co-simulation of multi-processor systems-on-chip. In *Proceedings 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, pages 494–499, 2002.
- [2] M. Besana and M. Borgatti. Application mapping to a hardware platform through automated code generation targeting a RTOS: A design case study. In *Proceeding of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 41–44, March 2003.
- [3] W. Cesario, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, and M. Diaz-Navia. Multiprocessor SoC platforms: A component-based design approach. *IEEE Design & Test of Computers*, 19(6):52–63, 2002.
- [4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.
- [5] A. Gerstlauer, H. Yu, and D. Gajski. RTOS modeling for system level design. In *Proceeding of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 130–135, 2003.

- [6] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [7] A. Haverinen, M. Leclercq, N. Weyrich, and D. Wingard. White paper for systemc based SoC communication modeling for the OCP protocol. [www.ocpip.org](http://www.ocpip.org), 2002.
- [8] F. Herrera, H. Posadas, P. Sánchez, and E. Villar. Systematic embedded software generation from SystemC. In *Proceeding of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 142–147, 2003.
- [9] S. Holloway, D. Long, and A. Fitch. From algorithm to SoC with SystemC and Cocentric System Studio. Synopsys Users Group (SNUG), 2002.
- [10] Jean J. Labrosse. *MicroC/OS-II, The Real-Time Kernel, Second Edition*. CMP Books, 2002.
- [11] Ian Mackintosh. Open core protocol international partnership. [www.ocpip.org](http://www.ocpip.org), 2003.
- [12] OSCI. SystemC version 2.0.1 User's Guide. [www.systemc.org](http://www.systemc.org), 2002.
- [13] I. Oussorov, W. Raab, U. Hachmann, and A. Kravtsov. Integration of instruction set simulators into SystemC high level models. In *Proceedings Euromicro Symposium on Digital System Design (DSD'2002)*, pages 126–129, 2002.
- [14] P. G. Paulin, C. Pilkington, and E. Bensoudane. Stepnp: A system-level exploration platform for network processors. *IEEE Design & Test of Computers*, 19(6):17–26, 2002.
- [15] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings Asia and South Pacific Design Automation Conference (ASP-DAC 2000)*, pages 405–408, 2000.

# Chapter 7

## LAERTE++: AN OBJECT ORIENTED HIGH-LEVEL TPG FOR SYSTEMC DESIGNS \*

Alessandro Fin, Franco Fummi

*Dipartimento di Informatica, Università di Verona, ITALY*

{fin, fummi}@sci.univr.it

**Abstract** This paper describes Laerte++, a high-level test pattern generator (TPG) for SystemC designs. All necessary features of a high-level TPG (e.g., fault models definition, hierarchical analysis, coverage measurements, etc.) are implemented by exploiting native SystemC characteristics funded on OO principles. The framework robustness and extensibility are guaranteed by an accurate use of software engineering methodologies for the Larte++ classes definition and an extensive use of the Standard Template Library (STL) for data structure definition. Laerte++ allows to set up and run an *ex-novo* TPG session by adding very few C++ code lines to any SystemC design under test description. The applicability and the efficiency of the presented framework have been confirmed by the analyzed benchmarks.

### 1. Introduction

The complexity of modern designs requires to approach testing problems from the early stages of the design flow. Testing activities performed at high abstraction levels are usually cited as high-level testing [1]. Such activities are based on some techniques and methodologies mixing concepts derived from both the verification and the testing fields.

A high-level test pattern generator (TPG) tries to generate test patterns by exploiting high-level fault models on high-level system descriptions, usually modeled by means of a hardware description language

\*Research activity partially supported by the IST-2001-34607 SYMBAD European Project.

(HDL), such as Verilog, VHDL and SystemC. High-level (functional) test patterns can be used, between others, for the following tasks:

- Comparison of a specification with an implementation [2], where patterns stimulate an implementation in comparison with a golden reference model to identify discrepancies.
- Design error identification [9], where the identification of redundant or hard to detect high-level faults is put in relation to symptoms of design errors.
- Testability analysis [5], based on the assumption that hard to detect high-level faults identify design corner cases, which will require the use of design for testability techniques at lower abstraction levels.
- Hierarchical test generation [7], based on the inheritance of test patterns from all steps of a design flow in order to simplify the TPG phase at the gate level.

Let us compare high-level TPG's by considering the way they interact with the design under test (DUT) modeled by using a HDL. Particularly, the way the TPG performs the following basic tasks, which are common to all high-level TPG's. *Fault injection* and fault simulation to compare a fault-free behavior to the faulty one; *coverage measurements* to guide the TPG engine; *hierarchical analysis* to move the focus of the TPG engine from one component to a sub-component and vice versa. Such basic tasks are implemented by using one of the following strategy to connect the high-level TPG to the DUT:

- connection to *external HDL* simulators [18],[19], thus being independent on the selected HDL, but avoiding to be able to exploit language information necessary to effectively guide the TPG engine;
- *code transformations* [4],[6],[12] requiring HDL parsing for extra modules insertion, this produces simulation degradation due to the alteration of the original DUT description;
- generation of an *internal model* of the DUT [8], that promises high performance and complete integration between the DUT and the TPG, but paying a complex and error-prone phase of HDL translation.

The proposed high-level TPG, so called *Laerte++*, mixes the best aspects of the second and the third strategy. It is focused on SystemC

DUT descriptions, which can be directly linked to the TPG code in order to produce a single executable code, which generates a TPG session every time it is executed. Note that, the internal model of Laerte++ is the SystemC language itself. Laerte++ exploits in depth the object oriented (OO) paradigm and some intrinsic characteristics of SystemC in order to implement the basic tasks of a high-level TPG. In this way no language transformations are necessary except for the injection of faults. This allows to sensibly improve performances and to simplify the set up of an *ex-novo* TPG session that is based on very few C++ code lines added to any SystemC DUT description. Moreover, the OO paradigm allows to easily extend Laerte++ with new fault models, TPG engines, coverage metrics, etc.

The paper introduces as first the Laerte++ philosophy, then its effective features are presented. Finally, a significant applicability example is analyzed.

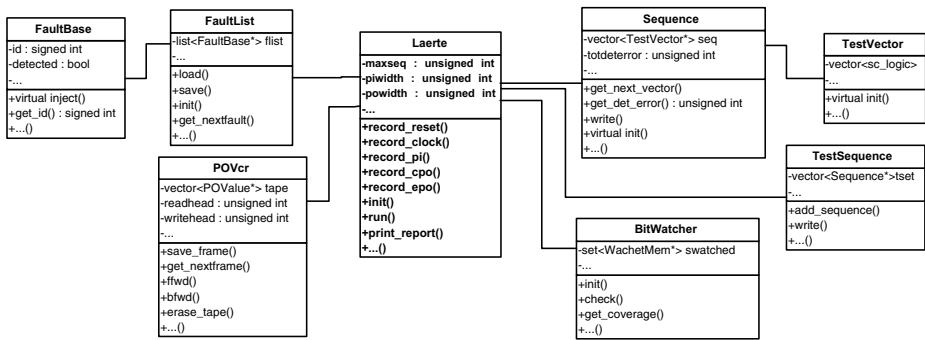


Figure 7.1. Laerte++ UML diagram.

## 2. Laerte++ Philosophy

The Laerte++ software architecture is deeply based on standard template library (STL) data containers [17] and native SystemC data types. The main **Laerte** class contains: a fault list manager (**FaultList** class), a set of TPG engines (**Sequence** classes), a test set (**TestSequence** class), a memory elements inspector (**Bitwatcher** class) and some classes for improving performance (e.g., **POVcr** class), see Figure 7.1. In this way, majority of software structures, necessary to implement a testing environment, are directly built by using SystemC language characteristics. For this reason, the amount of DUT specific C++ code required by

Laerte++ is less than 10% of the code required by the approach [12], which is based on a complete DUT code transformation.

## 2.1 Laerte++ architecture

Laerte++ has a five-layered architecture composed of:

**TPG engine.** It includes a set of C++ classes to generate test sequences for the DUT. Four test sequence generators have been defined and they can be dynamically selected run-time in order to apply the most effective engine to each target fault.

**Fault injector.** It is responsible for generating the faulty description of the DUT. This description is functional equivalent to the fault-free DUT, but it shows faulty behaviors by driving the added `injfault` port [12] to a value in the range from 1 to the number of injected faults.

**Input/Output Interface.** It includes all the modules to load/save test sequences, fault lists and test sets.

**Communication interface.** Laerte++ can establish a socket connection to remote SystemC simulation, in order to perform a remote test pattern generation phase [10]. Additionally, its communication interface is available to drive a *state of the art* hardware emulator, in order to burst testing performances.

**Graphic user interface.** Simulation and testing parameters can be set by both graphic user interface and command line.

## 2.2 Testing procedure set-up

The required effort for defining an *ex-novo* testbench should be considered as an important parameter to evaluate the effectiveness of a testing environment. Figure 7.2 shows how to define in Laerte++ a complete testbench for a DUT. The testbench definition for the considered example requires very few additional C++ code lines, which are so simple to be almost self explaining: the DUT is instantiated, as usual in SystemC, then names of its input-output ports are passed to Laerte++ with some other information related to the clock and reset signals, finally a test pattern generation session is activated. The compilation of the main code produces a single executable program. Its execution can be fully customized via either command line or GUI, in order to execute different TPG sessions.

```

#include "laerte.h"
#include "dut.h"
void main(int ac, char *av[])
{
    sc signal< bool > reset;
    sc signal< bool > dutclk;
    sc signal< int > injfault;
    Lrt signal<sc bv<2> > mode(2);
    Lrt signal<sc lv<8> > datain(8);
    Lrt signal<sc lv<4> > ctrlin(4);
    Lrt signal<sc bv<8> > dataout(8);
    Lrt signal<sc bv<4> > flagout(4);
    // DUT instantiation
    DUT *cdut;
    cdut = new DUT ("dut");
    cdut->reset(reset);
    cdut->clock(dutclk);
    cdut->mode(mode);
    cdut->datain(datain);
    cdut->ctrlin(ctrlin);
    cdut->dataout(dataout);
    cdut->flagout(flagout);
    cdut->fault port(injfault);
}

// Command line parameters parsing
Laerte laerte(ac, av);
// Reset & Clock signal registration
laerte.record reset(&reset);
laerte.record clock(&dutclk);
// Fault signal registration
laerte.record fault(&injfault);
// Input/Output signals registration
laerte.record pi(&mode);
laerte.record pi(&datain);
laerte.record pi(&ctrlin);
laerte.record cpo(&dataout);
laerte.record cpo(&flagout);
// Laerte++ initialization
laerte.init();
// Simulation/Test execution
laerte.run();
// Final report print
laerte.print report(&cout);
}

```

Figure 7.2. Laerte++ testbench.

## 2.3 Additional features

**Remote simulation.** Laerte++ allows to define testbenches for remote DUT simulation via socket communication [10]. The simulation/testing procedure is performed by a master-slave architecture.

**Hardware emulator interface.** An application programmable interface (API) for one of the most advanced hardware emulator has been developed [11]. It allows to load the emulator memory board and to sample the DUT primary outputs from the emulator equipment.

**Features extension.** Both the SystemC language and Laerte++ are based on C++. This allows to extend their original functionality by exploiting the object oriented paradigm. The Laerte++ library is composed of a set of C++ classes (see their UML schema in Figure 7.1), which can be extended in order to add new features. They can be used as base classes for deriving new classes, thus defining new and more specific features. For example, a new TPG engine can be easily defined by deriving it from the base class **Sequence** and by implementing the virtual method *init*, responsible to generate a test sequence. Another related example is reported in the next section concerning fault models. Moreover, since Laerte++ is based on a set of abstract classes, externally defined classes (e.g., random distribution, BDD managers, constraint

solvers, etc.) can be simply added to extend the implemented features (e.g., TPG engine, front-end, fault model, etc.).

### 3. Fault Injector

Laerte++ generates test sequences for the SystemC DUT description, by targeting high-level faults injected into the DUT code. A C++ class for fault injection is included into Laerte++ for performing fault injection and generating the faulty description. The `FaultInjector` class of Larte++ parses the SystemC DUT code and produces a faulty DUT description with inserted code saboteurs [14]. They represent high-level faults modeled by the *bit-coverage* metric, which has been showed [6] to include all well known coverage criteria such as, statement coverage, condition coverage, branch coverage and a large part of path coverage.

Each injected fault has a unique *id*, which can be activated by driving an additional port added to the original input/output DUT entity [12].

```

class TransientFault : public FaultBase
{
public:
    void inject() {
        if (poisgen->next() > threshold) {
            apply_fault();
        }
    }

private:
    void apply_fault();

    PoissonGen* poisgen;
    void* faulttarget;
}

```

Figure 7.3. TransientFault class definition.

#### 3.1 Definition of new fault models

Laerte++ defines the pure abstract `FaultBase` class to represent a generic fault model. The `BitFault` class is based on such a class to implement the *bit-coverage* metric. New fault models can be simply defined by deriving them from the `Fault` base class: it is only necessary to implement the virtual method `inject`. For instance, let us imagine to define a complex transient fault model. It is sufficient to identify a C++ class for random generation (e.g., [15]) and to select a random

distribution (e.g., normal, Poisson, etc.) to define in which time frame the fault is active, this is sketched in Figure 7.3.

## 4. TPG Engine

The core of the TPG engine is the pure abstract class `Sequence`. It is the base class for all kinds of derived sequence classes, see Figure 7.4. In the current Laerte++ implementation, three subclasses have been derived:

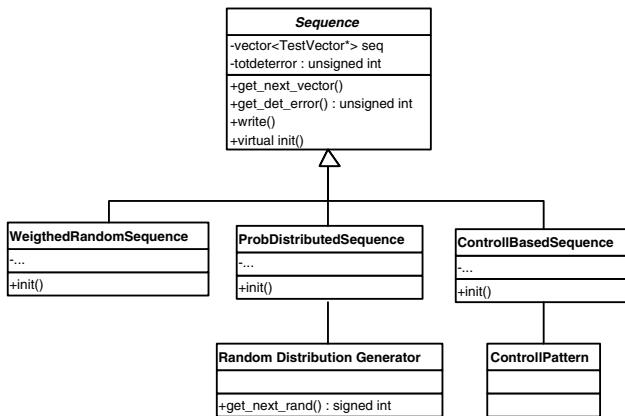


Figure 7.4. Sequence hierarchy.

**Weighted random sequences.** The virtual method `init` of the `Sequence` class is implemented to generate test sequences with a predefined percentage  $\alpha$  of bits set to 1 and  $(1 - \alpha)$  bits set to 0.

**Probabilistic distributed sequences.** Test sequences with the most appropriate distribution can be generated by including into the `ProbDistributedSequence` class a random number generator [15]. Such a random number library allows to define random permutations. This feature can be efficiently exploited for generating DUT highly tailored test sequences. For example, a CPU can be tested by generating opcodes with a required distribution (e.g., 40% of memory access, 30% of logic operations, 10% of arithmetic instructions and 20% of stack access). This allows to test the DUT with test sequences very similar to the software applications, which are executed on it.

**Control-based sequences.** This `Sequence` sub-class allows to explicitly define the value of the bits related to the control primary inputs

(PI) of the DUT, in the case a partition between control and data primary inputs can be done. Whenever control primary inputs are set to some constrained values, and data primary inputs are unspecified, the `init` method completes the unspecified bits of a test sequence by generating random values for the data primary inputs only, thus preserving control values. The `ControllBasedSequence` class allows to test the same functionality with alternative input data in order to deeply analyze specific design behaviors.

The C++ class inheritance paradigm allows to easily define new sequence classes by single or multiple class derivation. For instance, a control-based sequence with a Gaussian probabilistic distribution for the input data values can be obtained by deriving it from the `Controll-BasedSequence` class, but adding a reference to a Gaussian random generator. Figure 7.5 shows the source code sketching it.

```
class ControlledGaussianSequence : public ControllBasedSequence
{
public:
    void init() {
        ControllBasedSequence::init();
        InitDataBit();
    }

    void InitDataBit();
    ...

private:
    GaussianGen* gausgen;
}
```

*Figure 7.5.* A new sequence class definition.

Laerte++ simulation performance and testing effectiveness benefit of the following software components included into the `Laerte` class.

**Laerte++ signals.** Controllability and observability are key aspects to evaluate the testability of a sub-component of a DUT. To isolate a component of a DUT it is necessary to directly drive its input ports and directly read its output ports. Moreover, a hierarchical TPG must pass at run-time from the generation of test patterns for a module to the generation of test patterns for a set of modules or to the entire DUT. This could be possible by compiling different instances of the DUT where sub-components are isolated or by adding hardware port wrappers, (such as in [12]), which isolate or control modules by driving control signals of wrappers. Laerte++ efficiently solves the problem by defining the `Lrt_signal<>` class. It is derived from the standard

```

template <class T> class Lrt_signal : public sc_signal<T>
{
public:
    void write(const T& value) {
        if (!isolate) {
            sc_signal<T>::write(value);
        }
    }

    void control(const T& value) {
        sc_signal<T>::write(value);
    }

    const T& observe() const {
        return sc_signal<T>::read();
    }

    ...

private:
    bool isolate;
}

```

Figure 7.6. *Lrt\_signal* definition.

SystemC `sc_signal<>` template class. The `Lrt_signal` constructor has the default parameter `isolate` to select between the propagation of a signal value or the isolation of the connected port from the rest of the architecture. This behavior has been obtained by redefining the virtual method `void write(const T& value)` of the `sc_signal<>` class (see Figure 7.6). Moreover, two additional methods have been defined to control and to observe the signal values. Note that, signals can be isolated or controlled run-time by setting the `isolate` parameter. The solution based on `Lrt_signal` has several performance enhancements with respect to hardware port wrappers. No DUT architectural modifications are required and replacing any `sc_signal<>` with `Lrt_signal<>` can be implemented with a simple `define` instruction. Moreover, this solution guarantees to reduce the memory usage and the whole simulation time, (as showed in Table 7.1), in comparison with hardware port wrappers directly defined as SystemC components [12].

Table 7.1. *Lrt\_signal* improvements.

Memory	Time	# SC updates
-60%	-40%	-50%

**Primary output VCR.** A typical approach for fault simulation and test pattern generation is based on the comparison of the primary outputs (PO) of the faulty and fault-free DUT for each test sequence and target fault. It is well known that this produces a waste of elaboration resources, in fact, the fault-free behavior is identical during the application of the same test sequence. The overall memory occupation and the number of SystemC simulated events can be drastically reduced by exploiting the Laerte++ POVcr class. Only the faulty DUT is allocated and fault-free primary output values are stored once.

Table 7.2 summarizes the average performance improvements obtained by a testing procedure adopting the POVcr compared to the concurrent simulation and comparison of a faulty and a fault-free DUT.

Table 7.2. POVcr improvements.

Memory	Time	# SC updates
-	-75%	-50%

**Bit watcher and constraints.** Useful information for testing purpose can be obtained by measuring the DUT activation rate due to the applied test sequences. This measurement is performed by the `RegWatcher` class. A `RegWatcher` object can be defined for every DUT memory element, and it verifies after each clock cycle if the watched value has changed. Additional methods have been defined to extract the change rate of the watched memory elements. Such information are commonly used by TPG's to guide test generation to these states which have been insufficiently visited. Figure 7.7 shows how to declare a `RegWatcher` object for the `irreg` DUT memory element (first declaration). Moreover, it is possible to define constraints on the watched value (second declaration). Laerte++ checks for constraints satisfiability at each clock cicle. Violated constraints can either stop the simulation/testing procedure or generate a constraint violation report at the end of the TPG session.

```
RegWatcher<sc_int <3> > reg (&(dut->irreg),string("IR reg"));
reg.add_constraint(new LtCons<sc_int<3> >((void*) new int(3)));
laerte.add_watchreg(&reg);
```

Figure 7.7. A `RegWatcher` declaration.

**Clock and reset procedures.** Laerte++ defines default clock and reset procedures, which are applied to the DUT. The user can define the DUT specific procedures, whenever the default clock and reset procedures are

not adequate to the DUT complexity. This can be obtained by passing the function pointer to Laerte++ methods as in Figure 7.8.

```
void Laerte::record_user_reset (void (*rst_callback)());
void Laerte::record_user_clock (void (*rst_callback)());
```

Figure 7.8. Clock and reset customization.

The easy access of Laerte++ to internal DUT information is exemplified by the coverage results. Fault coverage, statement coverage and bitflip coverage are automatically computed for each TPG session.

## 5. Applicability Example

The complex inverse quantizer of the MPEG encoder [16] has been considered as benchmark for verifying the main capabilities of the Laerte++ framework. Figure 7.9 shows the DUT architecture.

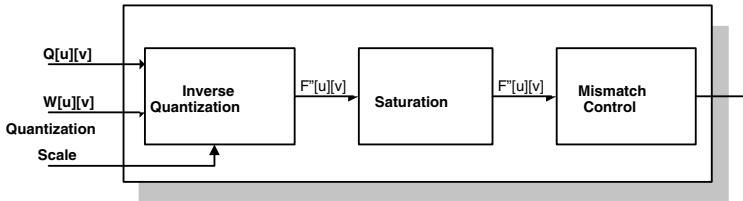


Figure 7.9. System architecture.

Table 7.3 shows the fault coverage and TPG characteristics for different DUT configurations. The first configuration tests the whole design. The Laerte++ testbench is less than 80 C++ code lines long. The obtained low coverage (59.1%) shows a quite low testability of the analyzed design. The time performances are improved by connecting the POVcr, which reduces by about 75% the TPG time and by 50% the number of SystemC updates. The overall coverage can be increased up to 69.9% by applying **ControlBasedSequences**. The control input values have been extracted by DUT functionality analysis. The majority of the undetected faults are located within the quantization module. By exploiting the controllability and observability capabilities offered by the Laerte++ signals, the inverse quantization module (*mod. Q*) has been isolated from the saturation and the mismatch control modules (*mod. S*). These macro submodules have been independently tested. As stated by the experimental results, the *mod. S* is not propagating most of the activated faults in the *mod. Q*.

Configuration	# Fault	Fault Cov.%	Time (sec.)	# Seq.	# Vect.	#SC update	Mem. Peak.
testbench(tb) only	3262	59.1	5405	47	30936	$> 10^6$	9.7 Mb
tb + vcr	3262	59.1	1488	47	30936	$< 0.5 \cdot 10^6$	8.8 Mb
tb + vcr + c/b seq.	3262	69.9	1863	68	49614	$< 0.5 \cdot 10^6$	8.8 Mb
tb + vcr + c/o (mod. Q)	2478	75.1	3965	36	23898	$< 0.3 \cdot 10^6$	7.2 Mb
tb + vcr + c/o (mod. S)	784	79.6	823	22	13324	$< 0.2 \cdot 10^6$	1.8 Mb
tb + vcr + c/o + c/b seq. (mod. Q)	2478	78.4	3878	39	25011	$< 0.3 \cdot 10^6$	7.1 Mb
tb + vcr + c/o + c/b seq. (mod. S)	784	83.2	792	19	11004	$< 0.2 \cdot 10^6$	1.9 Mb
c/b seq.: control-based sequence				c/o: Laerte++ signals			
mod. Q: inv. quantization submodule				mod. S: saturation & mismatch submodule			

Table 7.3. Test generation results.

## 6. Concluding Remarks

A high-level testing environment for SystemC designs has been presented. Object oriented principles and SystemC characteristics have been deeply used to simplify set-up and run of TPG sessions. Basic tasks of any high-level TPG have been implemented in the Laerte++ testing framework by extending standard SystemC classes. This avoids code transformations (as proposed in other high-level testing frameworks) that are error prone and impact on simulation performance. Experimental results show the efficiency of the proposed high-level TPG, which can easily analyze the high-level testability of a complex SystemC design. Future work will concentrate on the definition of fault models as SystemC classes extensions and the embedding of further test generation engines (GAs, SAT, etc.).

## References

- [1] V.M. Vedula, and J.A. Abraham. FACTOR: a hierarchical methodology for functional test generation and testability analysis IEEE Proc. DATE, pp. 730-734, 2002.
- [2] R. Leveugle Automatic modifications of high level VHDL descriptions for fault detection or tolerance. IEEE Proc. DATE, pp. 837-841, 2002.
- [3] S. Ravi, G. Lakshminarayana, and N.K. Jha. High-level test compaction techniques IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol.: 21 issue: 7, pp. 827 -841, July 2002.
- [4] L. Berrojo, I. Gonzalez, F. Corno, M.S. Reorda, G. Squillero, L. Entrena, and C. Lopez. New techniques for speeding-up fault-injection campaigns.

- [5] M.B. Santos, F.M. Goncalves, I.C. Teixeira, J.P. Teixeira. Implicit functionality and multiple branch coverage (IFMB): a testability metric for RT-level. IEEE Proc. ITC. 377-385, 2001.
- [6] F. Ferrandi, F. Fummi, D. Sciuto. Test generation and testability alternatives exploration of critical algorithms for embedded applications. IEEE Trans. on Computers , volume: 51 issue: 2, pp. 200-215, Feb. 2002
- [7] R.S. Tupuri, J.A. Abraham, and D.G. Saab. Hierarchical test generation for systems on a chip. IEEE Proc. ICVD, pp. 198-203, 2000.
- [8] R.B. Jones, J.W. O'Leary, C.-J.H. Seger, M.D. Aagaard, M.D., and T.F. Melham. Practical formal verification in microprocessor design. IEEE Design & Test of Computers , volume: 18 issue: 4, pp. 16 -25, July-Aug. 2001
- [9] A. Veneris, J.B. Liu, M. Amiri, and M.S. Abadir. Incremental diagnosis and correction of multiple faults and errors. IEEE Proc. DATE, pp. 716-721, 2002.
- [10] A. Fin, and F. Fummi, A Web-CAD Methodology for IP-Core Analysis and Simulation, IEEE Proc. DAC, pp. 597–600, 2000.
- [11] A. Castelnuovo, A. Fin, F. Fummi, and F. Sforza. Emulation-based Design Errors Identification. IEEE Proc. DFT, 2002.
- [12] A. Fin, F. Fummi, and G. Pravadelli. AMLETO: A Multi-language Environment for Functional Test Generation. IEEE Proc. ITC, pp. 821-829, 2001.
- [13] F. Ferrandi, A. Fin, F. Fummi, and D. Sciuto. Functional Test Generation for Behaviorally Sequential Models. IEEE Proc. DATE, pp. 403–410, 2001.
- [14] J.C. Baraza, J. Gracia, D. Gil, and P.J. Gil A Prototype of A VHDL-Based Fault Injection Tool IEEE Proc. DFT, pp. 396–404, 2000.
- [15] R. Davies Newran02B - a random number generator library <http://www.robertnz.net>
- [16] CoCentric SystemC Compiler Behavioral Modeling Guide. *Synopsys version 2000.11-SCC1*, 2001.
- [17] <http://www.sgi.com/tech/stl/>
- [18] <http://www.synopsys.com/>
- [19] <http://www.verisity.com/>

*This page intentionally left blank*

## Chapter 8

# A CASE STUDY: SYSTEMC-BASED DESIGN OF AN INDUSTRIAL EXPOSURE CONTROL UNIT \*

Axel G. Braun<sup>1</sup>, Thorsten Schubert<sup>3</sup>, Martin Stark<sup>2</sup>, Karsten Haug<sup>2</sup>, Joachim Gerlach<sup>2</sup>, Wolfgang Rosenstiel<sup>1</sup>

<sup>1</sup>*University of Tübingen, Department of Computer Engineering, Germany*

<sup>2</sup>*Robert Bosch GmbH, Reutlingen and Leonberg, Germany*

<sup>3</sup>*OFFIS Research Laboratory, Oldenburg, Germany*

**Abstract** This article describes the modeling and refinement process of an industrial application from the automotive domain starting from a C description down to a cycle accurate SystemC model. The work was done within a co-operation between Robert Bosch GmbH, OFFIS Research Laboratory, and the University of Tübingen. The application was given by an exposure control unit from a video sensor system of a Bosch driver assistance application. The objective was to study the design flow, starting from an algorithmic C description down to synthesizable SystemC, which is the starting point for further design activities, e.g. hardware synthesis. The case study includes a comparison of the C and SystemC implementation, an analysis and a discussion of the refinement and implementation process. The fixed-point to integer data type conversion was found to be a critical task within the design process, and an automated solution is provided.

**Keywords:** SystemC, System-Level Design, Fixed-Point Data Types, Refinement

\*This work was partly supported by the BMBF under contract 01M 3049 C (MEDEA+ project SpeAC).

## 1. Introduction

The increasing complexity of today's and future electronic systems is one of the central problems that must be solved by modern design methodologies. A broad range of functionality that must be adaptable to market requirements in a very fast and flexible way, and thereby a decreasing time-to-market phase are only some of the most important issues. There are several promising approaches like system-level modeling, platform-based design, and IP re-use, to face these problems. The economic success of a product highly depends on the power and flexibility of the design flow and the design methodology. A very promising and popular approach for system-level modeling is SystemC [12], [4], [6].

SystemC is a C++-based system-level specification language that covers a broad range of abstraction levels. Since its introduction in 1999, the specification language SystemC is becoming more and more a de facto standard in the domain of system level design. Today, with the availability of release 2.0.1 [4], SystemC corresponds to an advanced and well-organized system specification language and methodology, which is on the step of entering the industrial world. More and more commercial tools are built around SystemC, and a market of SystemC IP products starts to grow. On the other hand, many of today's systems, which are designated to be implemented by a System-on-Chip (SoC) are initially specified in C/C++ language. This is because most of today's system architects are working with C or C++, and for this, many SoC design projects are starting from pure software implementations. In addition, important standardization bodies (e.g. IEEE or ISO) are specifying their standards in a C-based specification style, which makes C/C++ the natural starting point for an implementation of standardized algorithms. In the this article, the modeling and refinement process of a Bosch exposure control unit, starting from a C specification down to a cycle accurate SystemC model, is studied. This example is considered to be typical for a wide class of data-flow driven applications in the area of advanced automotive electronics. An important aspect (which corresponds to one of the key features of the SystemC methodology) is to keep the system specification executable during the whole refinement process. This allows to simulate the system at every stage of the design process.

On a very high (system) level of abstraction, algorithm design is usually based on floating-point data types. During the design process bit widths and accuracies are evaluated and fixed-point data types are introduced. The introduction and evaluation process of fixed-point data types are well-supported by tools like FRIDGE (Fixed-Point Programming

and Design Environment) [14, 15] or Synopsys CoCentric Fixed-Point Designer [11]. SystemC offers a very powerful concept of fixed-point data types [4], but up to now these types are not directly synthesizable to hardware. In common hardware design flows an additional manual conversion has to be applied before the synthesis process. Fixed-point data types are transformed to an integer-based data format containing virtual decimal points (which correspond to binary points in the digital world). In our case study this conversion was found to be a very time-consuming and error-prone process. To close this gap, a methodology and a tool for the conversion of those data types and arithmetics to a synthesizable integer data format were developed.

The rest of the article is organized as follows: Section 8.2 describes the benchmark design given by an industrial exposure control unit. Section 8.3 presents the SystemC modeling and refinement process. Section 8.4 takes a detailed look at the fixed-point to integer data type conversion step. Experimental results are finally presented in 8.5.

## 2. Exposure Control Unit

Sensor-based driver assistance systems are becoming more and more important in the area of automotive electronics. Driver assistance systems support the driver in adequately reacting to dangerous driving situations. They prevent accidents or minimize their impact. For this task, several specialized sensor systems are combined, which cover different areas surrounding the car, ranging from ultra short distance parking assistance systems to ultra long distance sensors for adaptive cruise control (ACC). Figure 8.1 shows an overview on automotive sensor systems for driver assistance. In our investigations, we focus on a video-based vision sensor system which assists the driver by automatically observing and identifying objects (lane markings, other road users, traffic signs, ...) that are relevant for driving decisions. Therefore, the video system has to extract these objects from its actual video frame. This includes the detection of characteristic shapes and patterns (e.g., traffic signs), the identification of the position of other cars, and the calculation of distances to them. A very important task is the adaptation of the camera parameters to the actual illumination conditions of the car environment (exposure control) to guarantee a video frame with rich contrast. This object information can be used to warn the driver or to automatically interact with other active safety components of the car (e.g., for an automatic reduction of car speed). It's obvious to see that all these activities have to be done in real-time. Due to their huge data rates and high complexity, video sensors require a lot of calculating power and lead

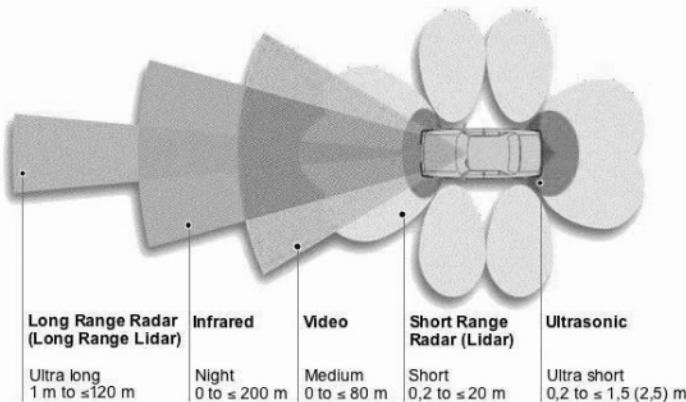


Figure 8.1. Automotive sensor systems

to complex systems with several processors and dedicated hardware to cope with these demands (see figure 8.2). Its processing part is typical

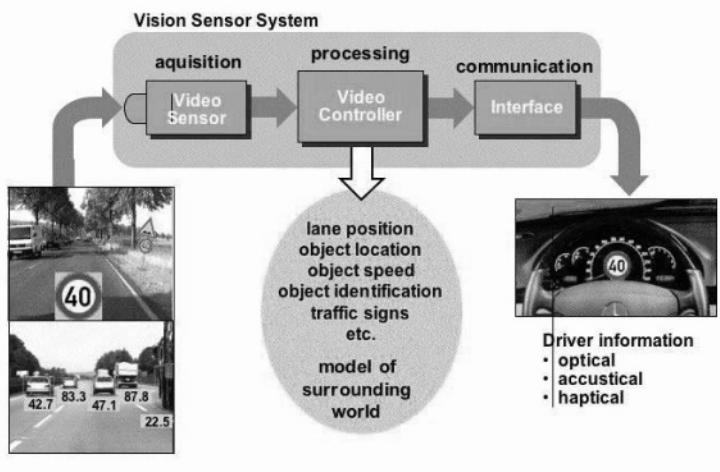


Figure 8.2. Vision sensor system

for the calculations to be performed in a vision sensor system. The exposure control unit (for short, ECU) calculates the exposure parameters gain, offset and integration time for the digital image sensor [3]. These exposure parameters are used to adapt the sensor to fast changing il-

lumination conditions during a car ride. The driver assistance system needs input image frames in a steady quality, even if the car is driving from bright sunlight into a dark tunnel for example. The data for a proper adoption of the image sensor is provided by the ECU. The ECU is connected to a VGA digital image sensor and consists of four functional units. The camera provides  $512 \times 256$  pixel images in 8 bit gray scales. The sample rate is 12 to 15 MHz. The first unit takes an image frame and calculates a 256-bit histogram, resolved into 16 bits. The histogram data set represents the basis for the following threshold calculation. The threshold values are calculated by accumulating the histogram values until different threshold areas are reached. These threshold values and the border values of the histogram data set are taken into the control unit for the calculation of the parameters gain, offset and integration time. The parameters are looped back to the digital image sensor via its serial  $I^2C$  interface [7]. To satisfy the hard real time constraints, the parameters have to be available for the next image frame at the digital image sensor. Figure 8.3 shows a block diagram of the entire system.

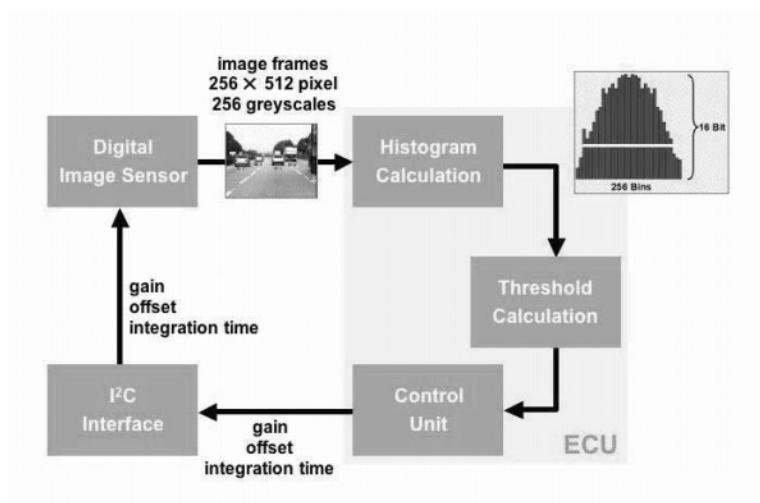


Figure 8.3. Block model of the ECU

### 3. SystemC Modeling and Refinement Process

In our investigations, modeling and refinement is done for the ECU using SystemC. Starting point for the modeling is an algorithmic software implementation given in ANSI-C. This model is structured by different C functions: The histogram generation, threshold calculation and the

control routine are separated in different C functions. The test data stream is read-in from disk files within the main function and supplied sequentially to the calculation routines. The final results as well as intermediate results are saved to disk in the main function. The test data for the original model consists of an image frame stream with about 1500 single image files and a set of data files containing the intermediate results. The images are partly generated from a real-world camera ride and partly generated artificially for special test cases, e.g. one half of a frame is black and the other one is white to test contrast behavior of the control routine. The reference data files contain all results of the histogram generation, the threshold calculation and the final results. This results in a scenario where the camera part of the vision system is replaced by a simulation model. The image stream, the intermediate data and the final results are used for all models developed during the refinement process.

The initial C model of the ECU was initially transformed in a system-level SystemC description and stepwise refined down to a cycle-accurate SystemC model for hardware synthesis. Several snapshots of the refinement process, that document the transformation process from an abstract system-level model down to a synthesizable lower level presentation, were discussed in detail in the following sections. Figure 8.4 shows an overview of the refinement process.

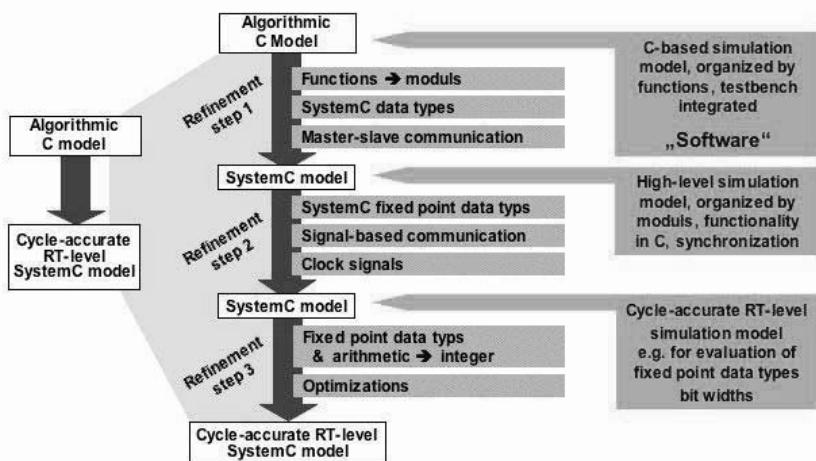


Figure 8.4. SystemC refinement process

**Refinement Step 1.** The first SystemC model is based on the SystemC Master-Slave Communication Library [5]. The model of computation provided by this library is well-suited to describe an abstract prototype close to the original software version. The functional blocks of the software version can be converted to SystemC modules easily. The specification of the behavior remains unchanged. The connection of the image grabbing is implemented by a FIFO. The FIFO de-couples the different clock frequencies of the camera and the image processing. The histogram unit and the threshold calculation unit are connected by a dual-ported RAM. The dual-ported RAM provides two memory areas. This ensures that the histogram unit can proceed with random access to the different values during calculation. The histogram unit can fill one area, while the other area is accessed by the threshold unit. Afterwards the memory areas for both units are swapped for the next cycle. The connections, which are implemented in the algorithmic software prototype by function calls, are replaced by the Master-Slave Libraries communication channels. Figure 8.5 shows a small part of the top-level code of the first model. The complete communication mechanisms (e.g. handshake communication or bus protocols) are hidden in the `sc_link_mp<>` channels. The camera and FIFO modules exchange data using these channels.

```

int sc_main(int argc, char *argv[])
{
    sc_link_mp<int> ch1; // channel from camera to fifo input
    sc_link_mp<int> ch2; // channel form fifo output to histo unit
    ...
    // camera
    camera camera1("Camera");
    camera1.signal_out(ch1);
    ...
    camera1.MCLK(cam_clk);
    ...
    // fifo block
    fifo f1("fifo");
    f1.wclk(cam_clk);
    f1.rclk(sys_clk_1);
    f1.Pwrite(ch1); // channel from camera unit
    f1.Pread(ch2); // channel to histo unit
    ...
}

```

Figure 8.5. Master-slave library communication

**Refinement Step 2.** A second SystemC model is implemented by introducing signal-based communication and replacing floating-point data types by fixed-point representations. The exchange of data between different modules is synchronized by handshake protocols. The transition from master-slave-based communication and sequential execution model to signal-based communication requires an explicit implementation of the synchronization mechanisms (in the previous model of refinement step 1, those mechanisms were provided by the master-slave channels as a built-in feature). Figure 8.6 shows the refined communication between the threshold unit and the dual-ported RAM (consisting of explicit request, acknowledge, read/write, and data signals). The threshold module reads histogram data out of the dual-ported RAM by initiating the request, and specifying the appropriate RAM address. An additional signal (not shown in the code in figure 8.6) ensures the selection of the right memory area. Usually, an embedded system target platform does

```
// connection from dual ported ram to threshold unit
sc_signal<sc_uint<16> > ch4_d;
sc_signal<int>          ch4_addr;
sc_signal<bool>         ch4_req;
sc_signal<bool>         ch4_ack;
sc_signal<bool>         ch4_nRW;
...
// ----- dual ported ram block -----
dp_ram dp_ram1("DP_RAM1");
dp_ram1.ram_port_a_d(ch3_d);
dp_ram1.ram_port_a_addr(ch3_addr);
dp_ram1.ram_port_a_req(ch3_req);
dp_ram1.ram_port_a_ack(ch3_ack);
dp_ram1.ram_port_a_nRW(ch3_nRW);
dp_ram1.reset_lo(reset_lo);
dp_ram1.reset_hi(reset_hi);
dp_ram1.ram_port_b_d(ch4_d);
dp_ram1.ram_port_b_addr(ch4_addr);
dp_ram1.ram_port_b_req(ch4_req);
dp_ram1.ram_port_b_ack(ch4_ack);
dp_ram1.ram_port_b_nRW(ch4_nRW);
dp_ram1.clk(sys_clk_1);
...
// ----- threshold unit -----
thresh thresh("THRESH");
thresh.histo_in_d(ch4_d);
thresh.histo_in_addr(ch4_addr);
thresh.histo_in_req(ch4_req);
thresh.histo_in_ack(ch4_ack);
thresh.histo_in_nRW(ch4_nRW);
...
```

Figure 8.6. Signal-based module communication

not support fast floating-point arithmetic. Therefore floating-point data types have to be replaced by fixed-point data types (and finally by an integer-based data type). Within the ECU example, only the control core is affected by this substitution. The determination of an appropriate bit width of the fixed-point data types was done by an analysis step, in which the values stored in a variable are observed. Due to a specification which is executable during the whole refinement process, this analysis step can be done easily by applying test data sets to the specification. As a result of the analysis step, a fixed-point data type version of the ECU was generated. The conversion step from floating-point to fixed-point data types is supported by several commercial tools. For the ECU, the conversion was done manually by introducing the SystemC `sc_fixed<>` and `sc_ufixed<>` data types which provides a fixed word length format with a fixed position of the binary point (see figure 8.7).

```

sc_fixed<22,10>    new_gain;
sc_ufixed<24,4>    old_gain;
sc_fixed<19,11>    Do;
sc_fixed<18,10>    Du;
sc_ufixed<8,1>     safety_margin;
sc_ufixed<10,9>    histo_width;

...
new_gain = old_gain*(sc_ufixed<8,1>)(1-safety_margin)*histo_width/(Do-Du);
...

```

Figure 8.7. Fixed-point data types

**Refinement Step 3.** Due to the fact that fixed-point data types simulate quite slow and they are not supported by the following design steps (e.g., synthesis to hardware [8]), the fixed-point data types have been replaced by an integer-based fixed-point representation containing a virtual binary point. This conversion is not supported by commercial tools and therefore needs to be done manually in a very time-consuming and error-prone process. Figure 8.7 shows a line of the control unit source code. The `new_gain` variable is of type `sc_fixed<22,10>`, a signed fixed-point type with 22 bits and 10 bits to the left of the binary point. The corresponding calculation involves five additional fixed-point types (the according type declarations are shown in figure 8.7). A re-implementation of these calculations using pure integer arithmetic has to align the virtual binary point according to the linked variables before the calculation step and re-align it afterwards. An approach for the conversion step is to align the data fields using shift operations. The entire calculation can be split into single operations and auxiliary variables. The shifting for constants can be replaced by pre-shifted constants, but

this introduces completely inexpressive bit patterns. Another approach is to use bit slices of variables. Figure 8.8 shows the calculation of the expression shown in figure 8.7 based on integer data types using shift and cast operations. The example shows that the conversion of fixed-point data type into integer data type increases code size significantly caused by cast and alignment operations. It is clear to see that a manual conversion makes the task a very time-consuming and extremely error-prone (plus hard-to-debug) process. In particular an exploration of bit widths for system optimization is extremely hard to manage. For an

```

...
sc_int<22> new_gain;
sc_uint<24> old_gain;
sc_uint<8> safety_margin;
sc_uint<10> histo_width;
sc_int<19> Do;
sc_int<18> Du;
...
new_gain = ( (sc_int<22>)(((sc_int<31>)((((sc_uint<39>
((sc_uint<29>)((((sc_uint<34>)(((sc_uint<24>)((((sc_uint<32>
(old_gain)) * (((sc_uint<32>)((1 - safety_margin)))) >> 7)))) * 
(((sc_uint<34>)(histo_width))))>>1))))<<8) / (((sc_int<39>)( Do -
(((sc_int<19>)(Du))) ))))))>>8 );
...

```

*Figure 8.8.* Integer-based calculation

automation of this step, a conversion tool developed at the University of Tübingen [1, 2] was used, as shown in the following section.

#### 4. Automated Fixed-Point to Integer Conversion

In our case study the conversion of fixed-point data types to integer data types was found to be the most critical part of the refinement process. Therefore an automated conversion tool [2] was developed at the University of Tübingen. The main features of the tool are the arithmetical correct application of conversion rules in a correctness-by-construction manner, the preservation of parts of the code, which contain no fixed-point data types and arithmetics, and preservation of code formating.

The conversion is structured into four different stages: Parsing of the SystemC code, scope analysis and data type extraction, analysis of all expressions in the code and the generation of the converted and adapted integer-based code. In the first stage the SystemC source code is analyzed syntactically and a complete parse tree is generated. The analysis within this stage cannot be restricted to SystemC keywords or to keywords related to the fixed-point data types. Besides SystemC-specific

syntax, the analysis must cover also C++ syntax [13]. The parse unit is able to handle context-free grammars. The grammar itself is combined of C++ and SystemC, but is specified separately, so it can be modified and extended very easily. The parse tree will be the major repository of all information collected in the following analysis steps. The second stage of the tool analyzes variable declarations and attributes. A very important issue is the scope of variables respective classes. The tool annotates these scopes in detail. Data types and the related scope information of variables or classes are the basis for the correct conversion. Both, data types and scope information are integrated into the nodes of the parse tree for the following processing stages. The third stage analyzes all expressions found in the parse tree by evaluating the data types and the scopes of the sub-expressions. The final type of a certain expression is evaluated by applying the related grammar rules and the data type information to the expression. The fourth stage of the tool replaces the fixed-point arithmetics code parts and inserts appropriate integer-based code (type cast and shift operations). Fixed-point variable declarations will be mapped to the appropriate SystemC integer data types.

The tool allows to evaluate and modify the fixed-point version of a model instead of the integer version. In the fixed-point version of a model, the designers can easily exercise full control over the bit-layout and parameters of fixed-point data types. Figure 8.9 shows a screenshot of the conversion tool, which interactively converts a fixed-point SystemC code (fragment) typed or pasted in by the user (left hand window) on-the-fly into a corresponding integer-based representation (right hand window). The tool covers all basic arithmetic operations. In addition a command line version for batch mode usage is available. The conversion output may directly act as an input for a SystemC synthesis tool, e.g. Synopsys CoCentric SystemC Compiler.

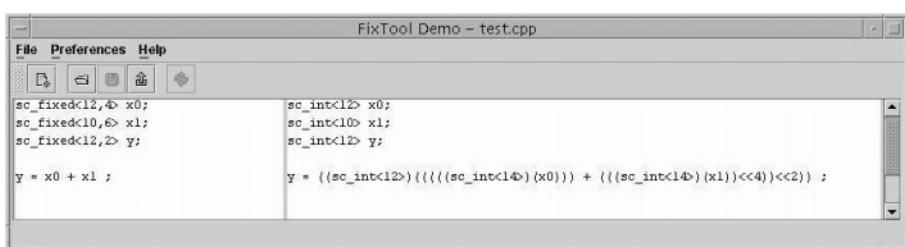


Figure 8.9. Automated fixed-point to integer data type conversion

## 5. Experimental Results and Experiences

In this chapter, the initial C version and refined SystemC versions of the ECU are compared in terms of code size and simulation speed, and the design experiences of the refinement and hardware synthesis process are summarized. SystemC requires some amount of lingual overhead to encode the specific SystemC coding style, this results in a slightly larger number of lines of code during the refinement process. On the other hand, this overhead makes a SystemC specification more structural organized and readable, and supports a well-defined refinement process. The binary code size increases by a factor of three during the step from C to SystemC, because the SystemC library includes the SystemC simulation kernel which is linked to the binary during the compilation process. So the increase of code sizes for both source code and binary code is an expected effect. Figure 8.10 shows a comparison of the execution (= simulation) times of the initial C specification and the SystemC models. For runtime benchmarking, the C version and the SystemC versions are stimulated by a set of approximately 1500 input image frames. All run time measurements were done on a 500 MHz SunBlade 100 machine. As

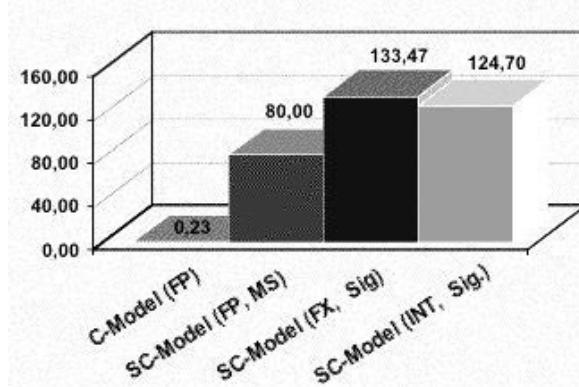


Figure 8.10. Execution times of C and SystemC ECU

figure 8.10 shows, the pure C-based software model (FP, floating-point) of the ECU is the fastest model. But this model contains no notion of time or clock cycles and includes no simulation kernel. Although it is fast (0.23 s/frame) it could not give any insight besides the basic functionality, because it provides no cycle accurate timing base. The introduction of communication (e.g. master-slave-based), in opposition to function calls in the pure C model, slows-down the simulation speed of the SystemC models significantly. In figure 8.10 the SystemC model with fixed-point types (FX) and master-slave communication (MS) shows this. Ad-

ditionally, the usage of SystemC fixed-point data types which simulate slower than standard SystemC data types (and particularly slower than C/C++ types) decreases simulation speed (refinement step 2, signal-based communication and FX data types). The gain of refinement step 3 (signal-based communication and integer-based types) is small due to additional adoption efforts (e.g. shift operations) for integer-based arithmetic. The integer-based SystemC model of the ECU was used for hardware synthesis. Therefore, particular blocks of the ECU were synthesized using a commercial available SystemC synthesis tool. The target hardware architecture was a Xilinx FPGA. Several ECU blocks, including the histogram and threshold unit, were successfully synthesized to the FPGA. The hardware implementations easily achieve the real-time requirements defined by the embedding environment.

Finally, some experiences from an algorithm designers point of view are summarized. The initial C specification is usually structured in different functions. These functions can easily be transferred into different SystemC modules. The core functionality can be re-used straight forward in a first step. The tight relation to C++ allows extending SystemC and its related design philosophy to the area of algorithm design. SystemC data types can be introduced during the refinement steps smoothly. C/C++ data types and SystemC data types can be used during conversion in parallel. The conversion of fixed-point data types to an integer-based data format is a time consuming and error prone task in the refinement process, so the availability of an automated tool is a big benefit (a strong requirement, respectively) at this point. The hardware synthesis tool applied determines the synthesizable subset of the SystemC specification language. Regarding the refinement process it is very important that this synthesizable subset is covered by the final refinement step, i.e. that the final model exclusively uses this subset. This allows to directly pipe the refined SystemC model into a hardware design flow.

## 6. Conclusion

This article shows the modeling and refinement process of an industrial application, an exposure control unit from the video part of an automotive driver assistance system. It starts from a high-level C description down to a ready-for-synthesis SystemC model. All transformation steps to be done during refinement are described. The conversion of fixed-point data types was found to be a basic problem, for which an automated solution was provided. The experiment shows that Sys-

temC provides a powerful methodology for a seamless validation and refinement process in an industrial design flow.

## References

- [1] A. G. Braun, J. Freuer, W. Rosenstiel. (2003). *An Automated Approach on Fixed-Point to Integer Data Type Conversion (Poster Presentation at SystemC University Booth)*, Design Automation and Test in Europe (DATE) 2003, Munich, March 3-7, 2003
- [2] A. G. Braun, J. Freuer, J. Gerlach, W. Rosenstiel. (2003). *Automated Conversion of SystemC Fixed-Point Data Types for Hardware Synthesis*. VLSI-SoC 2003, Darmstadt.
- [3] Motorola, Inc. (2000). *MCM20014, 1/3T Color VGA Digital Image Sensor, Semiconductor Technical Data*, MCM2001/D, 2000
- [4] Open SystemC Initiative (OSCI). (2001). *SystemC Version 2.0, Users Guide*
- [5] Open SystemC Initiative (OSCI). (2001). *SystemC Version 2.0, Beta-3: Master-Slave Communication Library*
- [6] Open SystemC Initiative (OSCI). <http://www.systemc.org>
- [7] Philips Semiconductors. (1998). *The I<sup>2</sup>C-Bus Specification, V2.0*
- [8] Synopsys, Inc. (2001). *Describing Synthesizable RTL in SystemC*
- [9] Synopsys, Inc. (2002). *CoCentric System Studio User Guide, Version 2002.05*
- [10] Synopsys, Inc. (2000). *CoCentric SystemC Compiler Behavioral Modeling Guide, Version 2000.11*
- [11] Synopsys, Inc. (2002). *CoCentric Fixed-Point Designer User Guide, Version 2002.05*
- [12] T. Groetker, S. Liao, G. Martin, S. Swan. (2002). *System Design with SystemC* Boston/Dordrecht/London: Kluwer Academic Publishers.
- [13] B. Stroustrup. (2000). *The C++ Programming Language (Special Edition)* Reading Mass. USA: Addison Wesley.
- [14] H. Keding, M. Coors, O. Luetje, H. Meyr. (2001). *Fast Bit-True Simulation*. 38. DesignAutomation Conference (DAC).
- [15] M. Coors, H. Keding, O. Luetje, H. Meyr. (2001). *Integer Code Generation for the TI TMS320C62X*. International Conference on Acoustics, Speech and Signal Processing (ICASSP).

## Chapter 9

# MODELING OF CSP, KPN AND SR SYSTEMS WITH SYSTEMC

Fernando Herrera, Pablo Sánchez, Eugenio Villar

*University of Cantabria  
Santander, Spain*

*fhererra,sanchez,villar@teisa.unican.es*

**Abstract** In this chapter we show the ability to specify with SystemC under the restrictions imposed by several model of computations, namely CSP, KPN and SR. Specifying under these MoCs provides some important properties, specially determinism and more protection against blocking, which are also important when implementation process is faced. In most cases, standard primitive SystemC channels or a combined use of them is suitable for specifying under these MoC restrictions. Nevertheless we provide some new specific and compatible channels providing important features, as dynamic checking of restrictions or atomic use. These channels represent an extension of the standard SystemC library.

**Keywords:** SystemC, system-level design, Model of Computation, SR, KPN, CSP.

## Introduction

Embedded systems are becoming prevalent in our daily lives as the electronic engines in a wide variety of appliances (telecom, home, automotive, medical, industrial, etc.). Their use should increase in the future with the advent of ambient intelligence [1]. Embedded systems will be implemented as Systems-on-Chip (SoC). The increasing complexity of SoC design requires new design paradigms [2].

SoCs are characterized by their heterogeneous nature. Heterogeneity affects both the architecture, which may be composed of different general-purpose processors, DSPs and application-specific HW modules, and the functionality, which may be composed of different, interactive functions. Depending on the characteristics, each function may be bet-

ter specified and designed using a specific Model of Computation (MoC) [3][4]. Therefore, in order to allow the designer to use the most appropriate specification method for each domain, the design methodology should support several models of computation. In this sense, Ptolemy [5] is a Java-based modeling and simulation environment able to support heterogeneous specifications composed of several domains, each characterized by a different MoC. In [4] a general study of the interfaces between different MoCs is made.

System specification is an essential step in any design methodology [4]. It constitutes the initial input to the design process. SystemC is gaining a wide acceptance by the users' community as system-level specification language [6][7]. SystemC is powerful enough to be a system-level specification language as it supports the basic features required for such an application [8][9]: programming constructs, object orientation (OO), FSM description capability, hierarchical, functional and structural partitioning, full system and environment description capability, concurrency and communication and synchronization mechanisms.

The main limitation of the language comes from the fact that it is based on a discrete-event, strict-timed computational model. This MoC has proven to be suitable for HW description at the RT level. Nevertheless, due to the abstraction provided by the OO capabilities of C/C++, the underlying, SystemC MoC can be hidden and other more abstract MoCs can be built on top [6]. This is shown in Figure 9.1:

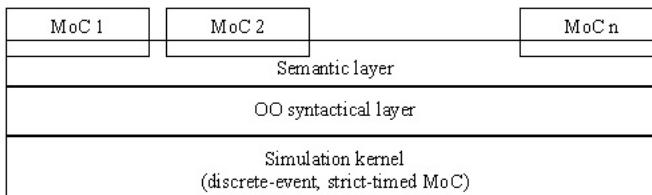


Figure 9.1. SystemC as a heterogeneous language able to support several MoCs.

Figure 9.1 has to be understood in the correct context. On the one hand, the discrete-event, strict-timed, underlying MoC limits the kind of abstract models that can be supported on top. So, for instance, continuous-time MoCs would require a different simulation kernel. On the other hand, relaxed-timed MoCs require timing abstraction. This is provided by  $\delta$ -cycles, for which a flexible interpretation has to be given.

In this chapter, a system-level specification methodology for SystemC will be presented. It follows an orthogonal separation between functionality and communication without limiting the specification capability

of the designer. This general-purpose specification methodology supports system-level performance analysis [10] and a design methodology following a single source approach, able to automatically obtain an implementation on the target platform [7].

As it is general, the specification methodology supports several computational models and even heterogeneous specification combining some of them. In this chapter, three classic, fundamental MoCs are studied. They are the Communicating Sequential Processes (CSP) [11], the Khan Process Networks (KPN) [12], and the Synchronous Reactive (SR) [13] MoCs. These models cover several application domains for which each one may be more suitable (i.e. process control, data processing or real-time reactive systems). All of them are supported by robust design methodologies for embedded systems guaranteeing the equivalence between the initial specification and the final implementation. So, for example, [14] addresses an efficient sequential implementation from a SR description written in Esterel. They also provide the conditions for formal verification. In the chapter, the application of the proposed specification methodology to these MoCs is presented.

The structure of the chapter is as follows. Section 9.1 describes the proposed, general-purpose SystemC specification methodology. In Section 9.2, the KPN, CSP and SR MoCs will be briefly described and the particularization of the specification methodology supporting these models will be presented. Finally, the main conclusions and future lines of work are given.

## 1. Embedded system specification in SystemC

### 1.1 Specification Structure

SystemC is proposed as a system-level specification language for generating the executable specification to be used as input to the HW/SW codesign process. The complete executable specification is divided into two different parts, the system specification and the environment model. There is no restriction in the way to model the system environment excepting in its communication with the system specification, which has to be done through ports and predefined channels. The general system specification structure is shown in Figure 9.2. The main elements of the system specification are the following:

**Process:** Composed of instruction sequences, which describe the concurrent functionality of the system. Both finite and cyclic are supported.

**Interface:** Class containing the set of methods without associated functionality corresponding to the different ways to access a channel.

**Channel:** Class implementing the methods of one or more interfaces for the communication among two or more processes.

**I/O Channel:** Channel for communication between the system and its environment. These channels are the boundaries for system design.

**Module:** Container class for the rest of the system specification objects. The modules allow hierarchical and structural partitioning.

**Port:** Module interfaces. Ports are the way modules communicate with each other and with the environment through channels.

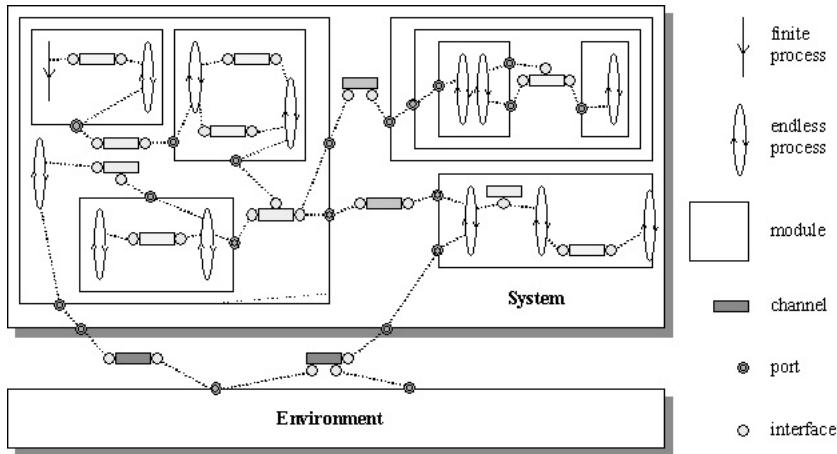


Figure 9.2. General system specification structure.

The system specification may contain IPs of any kind. The restrictions described in the following section do not apply to them as they do not take part of the co-design process directly. The only restriction is again communication with the rest of the system specification through ports. If an IP block does not satisfy that requirement, an appropriate interface has to be added.

## 1.2 System specification

**Structural and functional partitioning.** The main limitation imposed by the specification methodology proposed is a clear separation between communication and functionality. This ensures a reliable and efficient co-design flow. In order to achieve this, channels are the only way to communicate processes. When processes belong to different modules (module communication) this is done through ports. The process is the only way supported to describe concurrent functionality. Thus, func-

tional granularity is established at the process level. A module may contain as many processes as needed.

**Processes.** Only the SC\_THREAD, the most general process type in SystemC, is supported. All system processes are static, thus declared before system start and they are resumed with the sentence sc\_start() in the sc\_main() function. A process will stop execution when it reaches a timing wait or, eventually, a blocking access to a communication channel. No event object is supported, therefore, neither wait nor notify primitive are allowed.

**Communication.** Processes communicate among themselves through channels. Inside a channel it is possible to use wait and notify primitives. The designer is responsible for providing appropriate implementations for these communication channels on the implementation platform used. Primitive SystemC channels, such as ('sc\_signal', 'sc\_fifo', 'sc\_mutex', 'sc\_mq'. etc...), are supported provided that appropriate implementations are defined for them. In order to avoid the designer the need of developing new communication channels and interfaces, a communication library has been developed. The library is provided with implementation mechanisms independently of the final mapping of each of the processes involved to HW and/or SW resources. In addition to the library of channels for communication among processes inside the system specification, an I/O communication library has been developed. The aim of this library is to provide the designer with specific facilities for communication between the system and its environment. Again, when required, the designer should provide appropriate implementations for these channels over the platform used. In any case, the communication libraries can be extended with any new facility needed. The only restriction is that an appropriate implementation for the corresponding system-level object is provided.

**Timing abstraction.** As commented before, the underlying model of computation of SystemC is discrete-event and strict-timed with  $\delta$ -cycling. The tag 't' in any event  $e = (v, t)$  in any signal 's' has two components:

$$T(e) = t = (t_t, t_\delta) \in T \subseteq NxN$$

We use here the terminology in [3]. Thus, terms such as 'event' and 'signal' do not have the same meaning as in SystemC. Let us suppose two signals ' $s_1$ ' and ' $s_2$ ' generated by two concurrent processes  $P_1$  and  $P_2$  leading to the simulation results shown in Figure 9.3. Following the specification methodology proposed above, events in the strict-timed

axis ( $T_t$ ) will be generated by the environment ( $t_1$ ,  $t_3$  and  $t_4$ ) except for those generated by the execution of timed wait statements in any of the processes in the system ( $t_2$ ). The former will be called domain events. The latter will be called functional events. As this is a strict-timed axis, it is clear that:  $t_1 < t_2 < t_3 < t_4$ . In the general case (actually, this

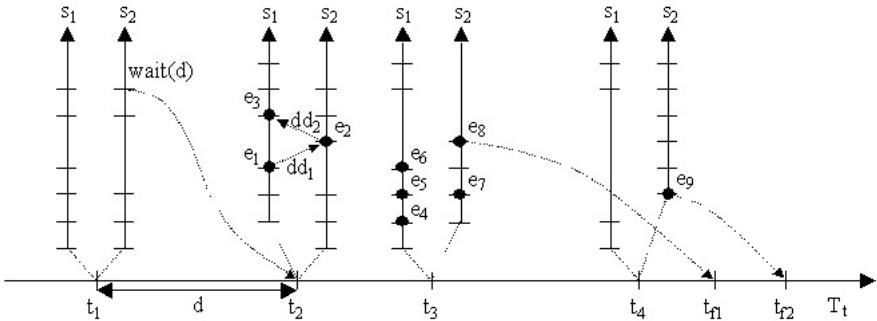


Figure 9.3. Simulation results.

can depend on the specific design methodology), domain events should remain invariant during the design process. This is not the case with functional events. So, for instance, the functional event ' $t_2$ ' does not have to be implemented exactly ' $d$ ' seconds after ' $t_1$ '. In each signal, the corresponding  $T_\delta$  axis is relaxed-timed and therefore, events are ordered. So, for instance, in signal ' $s_1$ ':  $T(e_4) < T(e_5) < T(e_6)$ . The  $T_\delta$  axis of different signals should have an untimed interpretation and, therefore, in general, the order among events between different signals does not have to be preserved. So, for instance, event ' $e_7$ ' can be scheduled, before, synchronous or after events ' $e_4$ ', ' $e_5$ ' or ' $e_6$ '. Only (control and/or data) dependencies can impose a certain ordering among events. So, for instance, as a consequence of dependencies ' $dd_1$ ' and ' $dd_2$ ', it is known that:  $T(e_1) < T(e_2) < T(e_3)$  and this ordering should be preserved during the design process.

The untimed model of the concurrent computations in the different processes and the relaxed-timed interpretation of the  $T_\delta$  axis of each signal are essential to provide the designer with enough freedom for design-space exploration to achieve an optimal implementation. The implementation should satisfy all the timing relations derived from the SystemC specification as well as all the timing constraints defined [9]. From this point of view, the design process becomes a function:

$$DP : T(SYS) \longrightarrow T_t$$

such that, for:  $\forall e \in s \in SYS \Rightarrow DP(T(e)) \in T_t$ . SYS being the system process containing all the signals in the system specification and T(SYS) all the tags in all the signals obtained from the SystemC specification. The design process is correct in terms of time when all the timing relations and constraints are satisfied by DP(T(SYS)).

## 2. Modeling of CSP, KPN and SR systems

The general code template of a system-level process is the following:

```
SC_MODULE(module_name) {
    void process_name( ) {
        // local initialiation
        while(true) {
            // cyclic behaviour
        }
    }
    SC_CTOR(module_name) { SC_THREAD(process_name); }
}
```

Cyclic behaviour will be composed of non-blocking code segments and channel accesses.

### 2.1 Modeling of CSP systems

A CSP system is a network of sequential processes communicating among themselves via rendezvous [11]. The system specification methodology proposed in the previous section can easily be adapted to this particular MoC by using only channels implementing the rendezvous. These channels are memory-less with blocking access at both ends. Each one of the processes involved finishes the channel access once the data transaction, if any, has been completed. Therefore, communication implies process synchronization.

Rendezvous between two processes can be implemented with two FIFOs of length 1 as shown in Figure 9.4. This emulation could even be extended for more complex rendezvous involving several processes. As a consequence, SystemC could support the CSP MoC just by using the standard 'sc\_fifo' channel. Nevertheless, this approach has the disadvantage of requiring static analysis for identifying the rendezvous and thus making clear the underlying CSP MoC. In addition, properties such as the number of calling processes cannot be checked dynamically. In order to overcome these difficulties, the communication library has been augmented with specific rendezvous channels. These channels will allow

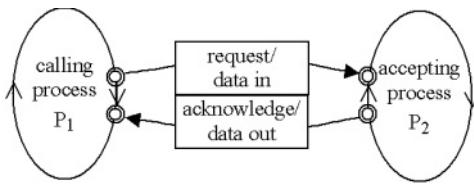


Figure 9.4. Rendezvous emulated with two FIFO channels.

the construction of a CSP specification and give additional flexibility for the specifier. The following channels are added to the library:

```
template<unsigned N>uc_rv_sync_channel: public uc_sync_if, ...
template<class T> uc_rv_uni_channel: public ...
template<class Tin=bool, class Tout= Tin>
uc_rv_channel: public uc_rv_if<Tin,Tout> ...
```

All of them are SystemC primitive channels (without internal structure). The first one, 'uc\_rv\_sync\_channel', is a very general synchronization channel. It has only an access method, 'sync()', thus, the channel is symmetric in the synchronization sense (there is no caller or accepter side). Any of the processes accessing this method will block until the N-th process arrival ('sync()' call). Then, the N processes will unblock together and the next N-1 arriving will block. N is configured at the channel instance. If each channel instance is accessed only by a set of N processes in the whole execution, determinism is preserved (otherwise, which of the N processes would take the rendezvous, would depend on the I/O and process speeds). A dynamic checking of this condition has been provided and can be activated in the channel. Dynamic checking is more general than static checking (done by means of ports in SystemC) because it enables the checking even for internal channels, where processes do not need access channels through ports. The checking can be disabled in order to gain simulation speed when the accomplishment of these conditions is ensured by other means (simplicity of the specification structure, repetition of simulations after a first one with checking, ...)

The other two channels, 'uc\_rv\_channel' and 'uc\_rv\_uni\_channel', present two additional features: they enable data transfer between processes involved in rendezvous and present an asymmetrical characteristic in the synchronization sense, that is, some processes act as caller processes, while others act as accepters. The notation caller and accepter refers only to synchronization and has nothing to do with the sense of data transfer. Basically, the difference is the flexibility in the number of pro-

cesses allowed for each type. Only one process will be able to act as accepter of the rendezvous channel instance. Due to the independence between channel and process in SystemC, the channel-process association forced by rendezvous is got again through dynamic checking (which, as before, can also be disabled). Once is activated, there are two possibilities with respect to the callers to a specific channel instance. By default, only one caller process will be allowed. If two or more processes access as callers an error will be raised stopping execution. With these conditions (one caller-one accepter process) determinism is ensured. In order to give flexibility, another possibility is to activate the SEVERAL\_CALLERS variable. This is applied over all the channel instances of the specification and means that several caller processes could access a rendezvous channel instance. The caller accessing each time in each rendezvous instance can vary (this being the source of indeterminism) provoking only a warning report at the end of execution. Another configuration option has been provided, CALLERS\_DETAILED\_REPORT, whose activation provokes the identification of how many and which processes accessed as callers for each specific channel instance. It should be pointed out that the specifier can still achieve determinism by introducing source information in transferred data (in band addressing).

While the characteristics shown above are common for 'uc\_rv\_channel' and 'uc\_rv\_uni\_channel', the main difference between them is their data transfer handling. The 'uc\_rv\_uni\_channel' implements a specific case which finds a direct association between synchronization and data transfer (perhaps semantically closer to the user). This channel uses directly a read/write style interface where the writer acts as caller (using a 'write()' or '=' operator access method) and the reader acts as accepter by calling 'read()' or '()' access methods. The transferred data type is generic and can be indicated at channel instance time.

The most general case of asymmetrical channel is given by 'uc\_rv\_channel'. This channel enables the different data transfer combinations (access methods in caller / accepter side):

```
void call(Tin &d_in, Tout &d_out) / void accept(Tin &d_in, Tout &d_out)
void call_write(Tin &data_in) / Tin& accept_read()
Tout& call_read() / void accept_write(Tout &data_in)
void call() / void accept()
```

For both, call and accept accesses, the meaning of 'data\_in' and 'data\_out' is the same and referred to the accepter, that is, 'data\_in' for data from caller to accepter and 'data\_out' for data from accepter to caller. The data type for each transfer sense can be different, although, this is not mandatory. The specifier can instance the channel with only a data type (then, both are used for each transfer) or without

type (then a Boolean type is assumed for both senses). Due to its greater generality, the ability to activate a calling correspondence for this rendezvous channel has been provided. When the variable CALLING\_CORRESPONDENCE\_CHECK is set, the channel dynamically verifies that each rendezvous makes compatible calls from each side, taking also into account the data transfer. The rendezvous allowed are of the type call(Tin,Tout)/accept(Tin,Tout), call\_write(Tin)/ acceptread(), call\_read()/ accept\_call(Tout) and call()/ accept(). If another combination were given, we would have an execution error, with a report of the type of call and accept and the channel instance.

As has been seen, a clear CSP specification in SystemC is feasible and provided the MoC is preserved by the implementation, the behavior and determinism will be guaranteed. Of course, the design process should be correct in time terms. The main problem of CSP systems is that they are prone to deadlock.

## 2.2 Modeling of KPN systems

A KPN system is a network of processes communicating among themselves via FIFOs [12]. The system specification methodology proposed in the previous section can easily be adapted to this particular MoC by using only FIFO channels. These channels contain a FIFO memory with non-blocking writing. Provided the FIFO is not empty, the reading is also non-blocking. If the FIFO is empty, the reading process will wait until new data are written. KPN implies asynchronous communication.

In order to support the KPN MoC the standard SystemC 'sc\_fifo' channel can be used [6]. Nevertheless, the channel library of the proposed specification methodology has been augmented with more complex FIFO channels:

```
template <class T>class uc_fifo_channel: ...
template <class T>class uc_fifo_inf_channel: ...
```

These are implemented in SystemC again as primitive channels with the same interface as the standard primitive SystemC 'sc\_fifo' channel, thus the same specification can make these interchangeable just by changing the channel instances. Both channels enable the transfer of a generic data type. The first channel, 'uc\_fifo\_channel', has the same internal implementation as the 'sc\_fifo\_channel', but including dynamic check of several conditions. In fact, the 'sc\_fifo' channel already includes a static check for determinism condition, allowing only the connection of one reader and one writer. As explained in the previous section, static checking has no effect when channels communicate internal processes inside a module without using ports. The 'uc\_fifo\_channel'

introduces dynamic checking for these conditions by means of the definition of the following variables: SEVERAL\_WRITERS\_ALLOWED and SEVERAL\_READERS\_ALLOWED. By default, for determinism conditions they would be undefined. These conditions are completely decoupled from the rendezvous channel conditions, that is, each type of channel has its own variables for activation of dynamic checking. If at some point of the execution a second writer or a second reader accesses to the same 'uc\_fifo\_channel' instance, then an execution error is raised (which differs from static check, which raises the error at binding time). If those variables are enabled, the execution will not provoke an execution error, but it will continue, reporting those 'uc\_fifo\_channel' instances accessed by more than one writer or more than one reader at the end of execution. In a similar way as in rendezvous channels, two variables: READERS\_DETAILED\_REPORT and WRITERS\_DETAILED\_REPORT enable the configuration of a detailed report where caller and reader processes for each channel instance with two or more read or write accesses are listed. Specifically, this channel allows another checking useful for detecting some blocking conditions by means of the variables: WRITER\_READER\_ALLOWED and READER\_WRITER\_ALLOWED. When these variables are activated, the dynamic checking also searches for those cases in which the same process performs first a write and then a read access or vice versa. Up to now, the 'uc\_fifo\_channel' covers some checking lacks of the 'sc\_fifo' by an increase of the simulation time.

There are two more problems that justify another SystemC implementation of a FIFO channel. Firstly, the pure KPN model of computation deals with infinite FIFO (thus unblocking write accesses), while the standard SystemC 'sc\_fifo' channel asks for a FIFO size (or gives a value by default). Secondly, a specification composed of 'sc\_fifo' channels, although more realistic and efficient, already assumes some previous knowledge of the system process speeds and internal token traffic in order to reduce blocking times. At system-level we do not actually have this knowledge until partition and performance analysis is done, which provides the optimal FIFO sizes.

Because of this, the 'uc\_fifo\_inf\_channel' has been provided. All the features mentioned for the 'uc\_fifo\_channel' are the same but, although it shares the same interface, the internal implementation of this channel is different from the previous one. In this case, none of the write methods ('write()' and 'write\_nb()') are blocking methods because of the infinite size of the channel. Of course, there is a real limit in simulation, given by the allocation limit of the host machine, where the simulation is run. Thus, the more powerful the host, the smaller the possibility of reaching this limit. If the specifier writes an endless generating process at

system-level (without time delays) this limit would be reached. In order to give some protection mechanism, a new parameter can be configured, DELTA LIMIT. If some value is given to this parameter, the channels will use this limit as the maximum read transfers and maximum write transfers of the channel in the same delta cycle. This variable is not required once time delays for performance analysis are introduced. This introduces one of the useful features of the channel, the 'uc\_fifo\_inf' channel can be configured by means of the variable MAX\_USED\_SIZE for reporting the maximum size used by each channel in the whole execution. This would allow an optimum FIFO dimensioning for the given run. Without time delays, this figures would give us the worst case FIFO sizes.

In order to forecast a real limitation of the platform, this channel admits another configuration variable, IMPLEMENTATION\_SIZE. If this variable is set, then the channel implementation includes a global (for all 'uc\_fifo\_channel\_inf' instances) monitor that maintains the maximum memory size required at each moment of execution. Then, the variable represents the maximum allocable memory (in bytes) for the FIFO channels in the final target. If the limit is reached, an execution error is raised, otherwise, the maximum memory size required for the channels is reported at the end of simulation. This figure is the sum of each 'uc\_fifo\_inf\_channel' instance maximum size (which considers the size in bytes of the data type transferred).

A KPN system is deterministic by construction and, provided the MoC is preserved by the implementation, the behavior will be guaranteed. Of course, the design process should be correct in terms of time. An additional advantage of KPN systems is that they are deadlock-free when the network lacks feedback loops. Nevertheless, KPN presents some limitations in modeling control systems, while, on the contrary, it allows the modeling of data processing systems in a very natural way.

## 2.3 Modeling of SR systems

SR systems handle concurrency under the 'perfect synchrony hypothesis', which states that each process in the system reacts to events in its inputs in no time [13]. As a consequence, the system is discrete-timed, that is, any event in any signal is synchronous to an event in any other signal and vice versa.

It could seem that limiting the proposed methodology with processes communicating through signals would directly implement the SR MoC by considering that all the events in the  $T_\delta$  axis of any signal are synchronous:

$$T(e_i) = (t_{ti}, t_{\delta i}) \equiv T(e_j) = (t_{tj}, t_{\delta j}) \iff t_{ti} = t_{tj}$$

Nevertheless, the SystemC signal is not appropriate as a transaction since a signal assignment with the same value as the current value of the signal does not create an 'event'. In order to overcome this problem, the 'sc\_buffer' can be used instead as the communication channel. The next example illustrates its use:

```

SC_MODULE(module_name) {
    void process_name( ) {
        // local initialalation
        while(true) {
            wait();
            if(ch1-> event() && ch2-> event()) {...}
            else if(ch1-> event()) {...ch1-> read();... }
            else if(ch2-> event()) {...ch3-> write(out_token);... }
            else { // not expected, sensitive only to ch1 and ch2}
        }
    }
    SC_CTOR(module_name) {
        SC_THREAD(process_name);
        sensitive << ch1;
        sensitive << ch2-> default_event; // both possible
    }
}

```

As can be seen, the description style is slightly different. In this way, the process has a sensitivity list where the synchronous channel ('sc\_buffer' in this case) instances that trigger its behaviour are specified. If, in a specific delta cycle, one or more of those channels ('ch1' or 'ch2' in the example) were written, then the functionality would be triggered. The 'event()' method enables the process to know which channels were written in the previous delta cycle, since, in the 'sc\_buffer' channel, each write provokes a delta event. During the execution, the channel value can be read as many times as desired. If the value read, nothing happens, but the values are missing for the next delta cycle. During execution, a write can be done (over ch3 in the example) that will trigger the execution of those processes sensitive to it.

A channel called 'uc\_SR\_channel' has also been provided. It is fully compatible with the 'sc\_buffer' interface (thus with the 'sc\_signal' interface) and adds a 'written()' method that can be used instead of 'event()' (which could be more expressive for the specifier). Nevertheless, its main contribution is that the channel can configure the dynamic checking of three main conditions activated by the definition of the following variables: CHECK\_WRITE\_BURST, CHECK\_OLD\_VALUE\_READ, CHECK\_LOST\_DATA.

The WRITE\_BURST check detects if more than one write (there being several writers or not) has been done in the same delta cycle. If this variable is not set, the last value written in the current delta will be read in the next delta, which would be a non-determinism source. This checking prevents several writer processes writing in the same delta, but does not detect when several processes do it in different deltas, therefore the SEVERAL\_CALLERS condition is still useful here. In a similar way, a check can detect if several processes react to the channel. The OLD\_VALUE\_READ check detects if a read has been done without a write in the previous delta cycle. This would detect the case of reading a channel that did not provoke process reaction, thus reading an old value. Finally, the LOST\_DATA check tests if a particular write was not attended to or inspected by the reacting process. It tests if the 'event()' or 'written()' method was not accessed in a channel that was written (thus triggering the process) in the previous cycle.

When a deterministic SR MoC is used, the implementation must fulfil the time environment restrictions and the design process DP has to ensure that all the  $\delta$ -transactions in one cycle finish before the next domain event:

$$\forall e = (t_i, t_{\delta j}) \in s \in SYS \Rightarrow DP(T(e)) < t_{i+1}$$

Then, the system-level MoC is preserved and the behavior will be guaranteed. An additional advantage of SR systems is that they can be made deadlock-free.

### 3. Conclusions

In this chapter, it has been shown that SystemC can be used in the specification of complex embedded systems under different MoCs. The proposed, system-level specification methodology is based on a clear separation between communication and computation. This characteristic is considered essential for system analysis and functional mapping. The three MoCs which have been analyzed are CSP, KPN and SR. All of them are classical, fundamental, well-known and understood MoCs for which efficient design methodologies exist.

Although the standard, primitive channels 'sc\_fifo' and 'sc\_buffer' could support the specification of KPN and SR systems respectively, additional, specific channels have been developed in order to provide the designer with two additional advantages. Firstly, to dynamically check the fulfillment or not of the fundamental conditions imposed by the MoC. Secondly, to provide the designer with additional description capabilities. In order to support a CSP MoC, rendezvous with delta synchronization have been provided by means of new specific rendezvous channels.

The timing flexibility in each case has been commented on. Understanding the temporal transformation allowed during the design process is essential in order to guarantee that the initial system specification and timing constraints are maintained. During system design, the fundamental assumptions of the MoC used at system-level have to be preserved. The general conditions for design correctness have been defined.

The two main research areas for further investigation are firstly, to study the SystemC specification of heterogeneous systems combining a mixture of MoCs in several interacting domains. In this sense, combination of timed and untimed MoCs is of particular interest. Secondly, the automatic analysis of the correctness of the design process.

## References

- [1] E. Aarts, and R. Roovers: IC Design challenges for Ambient Intelligence. Keynote Speech at DATE 03, available at <http://www.date-conference.com/conference/2003/keynotes/index.htm>.
- [2] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd: Surviving the SoC revolution: A guide to platform-based design. Kluwer; 1999.
- [3] E. Lee, and A. Sangiovanni-Vincentelli: A Framework for comparing Models of Computation. IEEE Trans. on CAD of ICs and Systems, V.17, N.12, December 1998.
- [4] A. Janst: Modeling Embedded Systems and SoC's , Morgan Kaufmann, 2003.
- [5] <http://ptolemy.eecs.berkeley.edu>
- [6] T. Grötker, S. Liao, G. Martin, and S. Swan: System Design with SystemC. Kluwer; 2002.
- [7] W. Müller, W. Rosenstiel, and J. Ruf: SystemC: Methodologies and Applications. Kluwer; 2003.
- [8] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong: Specification and Design of Embedded Systems. Prentice-Hall, 1994.
- [9] E. Villar: Design of HW/SW embedded systems. Servicio de Publicaciones de la Universidad de Cantabria, 2001.
- [10] F. Herrera, E. Villar, and F. Blasco: System-Level Dynamic Estimation of Time Performance for Codesign based on SystemC and HW/SW platform. Proc. of DCIS'02, Servicio de Publicaciones de la Universidad de Cantabria, 2002.
- [11] C.A.R. Hoare: Communicating Sequential Processes. Communications of the ACM, V.21, N.8, August 1978.

- [12] G. Kahn: The Semantics of a simple Language for Parallel Programming. Proc. of the IFIP Congress 74, North-Holland, 1974.
- [13] A. Benveniste, and G. Berry: The Synchronous Approach to Reactive and Real-Time Systems. Proc. of IEEE, V.79, N.9, Sept. 1991.
- [14] S.A. Edwards: Compiling Esterel into Sequential Code. Proc. of DAC'00, IEEE 2000.

# Chapter 10

## ON HARDWARE DESCRIPTION IN ECL

Lluís Ribas, Joaquín Saiz\*

*Microelectronics Unit – Computer Science Dept.*

*Autonomous University of Barcelona (UAB); Catalonia, Spain*<sup>†</sup>

Lluis.Ribas@uab.es

**Abstract** ECL is a system design language suitable for heterogeneous, reactive systems, but there are no tools to directly compile the data-flow parts into hardware. In this paper, we propose several approaches to synthesizing these parts into hardware without extending the language. As a consequence, the resulting environment enables a true HW/SW co-design for small and medium-size systems.

**Keywords:** System specification languages, hardware description languages, Esterel, ECL, C-based SLDL

### 1. Introduction

System-level design languages (SLDL) must enable designers to express a broad range of behaviors so to describe a number of systems that can include several components with distinct characteristics (communication protocols, analog components such AD/DA converters, data-flow processes, et cetera.)

As nowadays' hardware parts are able to perform complex tasks and they even include specialized functional units or embedded application-specific processors, they resemble processors that are used to execute software components. Therefore, preferred SLDL are inspired in programming languages, namely, C [8] or C++ [13]. (There are other SLDL based on other languages, too.)

\*Supported by an UAB researchers' grant.

<sup>†</sup>Partial funding provided by MCYT grant TIC2001-2508.

Of course, all SLDL do extend the original programming language they are based upon. SystemC [6], [7], [14], [15] takes profit of the object-oriented programming features to extend the base language with new operations and data types through the use of class libraries (one for each abstraction level and purpose).

On the other hand, other approaches extend the original language by new constructs supported by dedicated tools. For instance, HandelC [4] is a language based on a subset of ANSI-C with extensions to include parallel task definitions and communications via channels. However, it is not intended to fully cover the behavioral abstraction level. Rather, it is intended as a high-level abstraction description language for hardware. Also, SpecC [5] is defined as an extension to ANSI-C with constructs for implementing parallelism, synchronization, word-bit operation, and explicit FSM descriptions. (In HandelC, FSM are derived from the control-flow graph of the programs.) ECL [11, 10] belongs to this class of languages, as it is an extension to C language with constructs for inter-task communication and synchronization, and cycle-accurate semantics; which are derived from the Esterel [2] synchronous language. Also derived from it, there is no need to explicitly describe FSM.

Whatever the C-based SLDL, system descriptions are straightforwardly translated into programs that are, in fact, executable specifications. As a result, system simulations consist of specification executions. Likewise, software components are easily synthesized from specification. Consequently, their compilation into system processors' machine languages is a direct step in the system design flow.

System specification usually starts at a very high level of abstraction where a few components are described. The design process can be viewed as a chain of specification refinements in which each step consists of adding detailed information to proceed further on the design flow until "implementable specifications" are generated. Ideally, a single SLDL should be used all the design flow through. This simplifies designers' tasks and enables easy verification of each design stage (test benches from one design stage can be applied to the next, regardless of specific tests, which surely must be performed, too.)

As a result, any SLDL must be able to be used as a programming language or hardware description language (HDL), indistinctly. As for the latter, it must cope with RTL abstraction level constructs. At this level, EDA tools can be used to synthesize component descriptions targeted to hardware in a HW/SW system. The approach taken by SystemC requires using a dedicated class library and restricting specifications to the appropriate synthesizable subset of the language. Conversely, HandelC does not need such restrictions because of being, fundamentally,

a high-abstraction level HDL. In between the two extremes, there are SpecC and ECL.

ECL is not oriented to high-level, whole system specification. Instead, it is intended for medium abstraction level specification of control-dominated, reactive systems. However, being focused as such, their specifications are more controllable (i.e., cost estimation is more accurate) and easily refined into software programs and/or RTL descriptions for hardware. Furthermore, FSM do not need to be explicitly described and FSM constructivity problems (e.g., what would be translated in hardware as combinational feedback loops) are detected by the Esterel compiler.

Therefore, it is possible to use the formal background of Esterel to verify systems and a translation to C is readily available. In addition, ECL may be used as the entry language to POLIS [1] and, thus, to a co-design tool suite in which some system parts can be implemented in software and some others in hardware.

As a result, it looks like a good SLDL for mixed HW/SW systems. In fact, system simulation and software generation are straightforward. Hardware generation, however, requires more attention to corresponding specification.

The purpose of this paper is to investigate the hardware synthesis problem for ECL provided that verification and software synthesis pose no special difficulties. More precisely, to continue a previous work [12] in which a simple 8-bit processor was specified and partially synthesized. In that experiment, some bugs in the ECL compiler were detected and got around, but the data-path synthesis was shown to be difficult. Therefore, in this paper we propose a solution to the problem of data-path specification and synthesis in ECL; i.e., to extending the use of ECL to include some data-processing components. In the proposed approach, we take profit of the existing tools around ECL; i.e. Esterel [3] and POLIS, which includes a certain capability for synthesizing combinational arithmetic and logic modules.

The paper is structured as follows. After a short introduction to ECL, we shall briefly overview the co-design flow in the ECL environment and propose several approaches to an unrestricted hardware synthesis. In the next section, we shall describe the processor selected as a benchmark for the previous approaches and discuss about the results. The last section presents the conclusions of this work.

## 2. Overview of ECL

ECL stands for Esterel-C language and is, in fact, the result of adding Esterel constructs into C. As a result, the language looks like a C with statements to declare signals that are used for communication among modules, and with synchronization, preemption, as well as concurrency expressiveness capability.

The basic functional unit is the module, and each ECL description has a top-level main module. It is assumed that it is repeatedly called to compute. The calling may be done by a RTOS, or controlled by a global clock. Each time the module is called, it performs a computation by reading its current inputs, executing beginning in the previous halt point and continuing to the next one, and eventually emitting its outputs. Modules are hierarchically defined and can contain calls to C functions.

ECL modules are translated into either Esterel modules or C functions for further processing. However, all operators with no Esterel counterpart are translated into calls to C functions unless the affected module is translated fully into C. (It is, of course, interesting to keep the modules in Esterel, as it enables to choose between software and hardware implementations in the following design stages.)

All reactive statements from Esterel are present in ECL. Thus, it is possible to wait for a signal to be present (`await`), to check for its presence in the current instant (`present`), to emit a signal that will be available in the next time instant (`emit`), to simply wait for some number of instants to go away (`delay`), to include preemption among input signals (`do...abort`, `do...weak_abort`, `do...suspend`), and to parallelize the execution of some blocks of code (`fork {...} join`).

In the following, we shall describe the use of ECL in a HW/SW co-design environment and, particularly, its use as HDL.

## 3. HW/SW Co-Design Flow with ECL

ECL is a language with associated tools to translate it into C and/or Esterel. Therefore, each ECL component description can be converted into a C function or into an Esterel module, depending on the selected option for the ECL compiler. Eventually, Esterel modules contain calls to C functions to cope with operations that have no counterpart in Esterel.

From Esterel, there is a path to either C or VHDL [9] (through blif); i.e., to software or hardware. Furthermore, Esterel has been adopted as an entry language for components in the POLIS co-design tool suite. As a consequence, ECL can act as a single specification language for mixed HW/SW systems. However, translation of ECL modules into VHDL (or

any other hardware description language) is not as simple as generating software components.

The design flow previously outlined is shown in Fig. 10.1, which emphasizes that hardware parts can only be obtained from Esterel descriptions. That is, the output of an ECL compilation, whatever the result language is, can be verified by the Esterel simulator, but only those modules that are converted into Esterel modules are candidates to be further compiled into either a C program or an HDL.

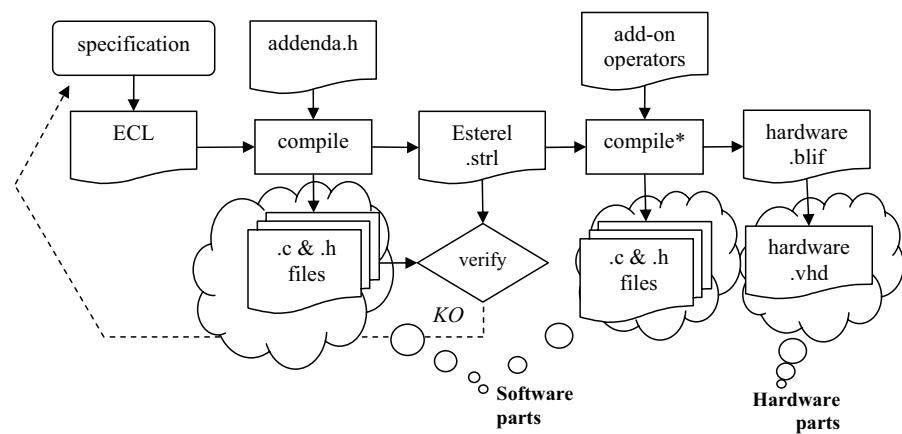


Figure 10.1. HW/SW co-design flow with ECL.

The fact that Esterel has tight restrictions when generating HDL descriptions (basically, only FSM can be synthesized to hardware) strongly affects the usability of ECL as an independent HW/SW co-design language, even though when used within the POLIS framework. In this section, we shall review the co-design flow and, especially, the hardware synthesis aspects of it.

### 3.1 Specification and Refinement

In a first approach, a functional, software model of the system is enough for a functional validation through simulation in Esterel, C-program or Ptolemy.

This first specification does not need to take into account possible system partitions (i.e., assignments of modules to either hardware or software). For the sake of simplicity, we have developed some complementary macros and C functions in a file included to the ECL package. (The designer has only to `#include ECLaddenda.h` in the specification

file.) Basically, it includes a Boolean data type definition to cope with eventual C-only compilations and several macros to improve readability to descriptions from which, `pause` (that is not implemented in the ECL, though it appears in the manual) and `loop {...}` are the most remarkable.

However, a functional, “free style” specification might require re-specifying some parts if targeted to hardware. Particularly, only those modules generated as “pure-Esterel” ones can be synthesized into hardware. A pure-Esterel module is a module that only receives and emits pure events; i.e., does not operate with typed data (Boolean or integer values.) Within POLIS, Esterel modules that do not call C functions can also be translated into hardware.

Therefore, once the system is functionally validated, a specification refinement must be done in order to enable HW-targeted modules to be generated as pure Esterel modules or, at least, as non-C dependent Esterel modules. (The modules targeted to software are readily converted into C functions.) To do so, the ECL compiler must be instructed to prioritize generation of Esterel instead of C in constructs that admit both translations. In short, once the validated system is partitioned (i.e., once it is known the destination –software or hardware– of each system component), HW-intended modules must be refined so that they can be totally translated into Esterel.

To empower the expressiveness of HW modules, the `ECLaddenda.h` file also contains logic, integer and bit-level operations that must be used instead of C-only operators. With this, a translation into Esterel-only descriptions is possible.

An important limitation in the specification of HW-intended modules is that there must be no data operation loops in their behavior; i.e., designers must restrict themselves to use Esterel constructs for waiting, concurrency and pre-emption and non-iterative instructions of C. (In fact that would be an RTL subset of ECL.)

In a HW/SW co-design environment, some modules can be re-partitioned into a different target. It is obvious that there is no problem in changing the implementation option of a HW module, but the opposite would surely require refining the module description. In the next subsection, we shall review all the options of hardware synthesis that are available from an ECL description.

### 3.2 Hardware Synthesis

In this section we shall consider the synthesis of a single ECL module that has been described following the rules previously described (i.e., limited to RTL-ECL with Boolean, bit, logic and integer operations.)

As introduced in the previous section, an ECL module must be translated into Esterel in order to be synthesized to hardware. The result Esterel code is then compiled into a reactive part (an extended finite state machine representing most of the ECL program), and a pure data part. The reactive part can be robustly estimated and synthesized to hardware, while the data part had to be originally implemented in software. That is, data-path synthesis requires some additional effort, which is partly done by enabling designers to specify integer-compatible data and operators.

Figure 10.2 shows the diversity of design paths to hardware from an ECL module that we had explored. If it has only pure signals it is possible to generate hardware description by directly using Esterel tools. In this case, only FSM can be generated. It is also possible to generate small, combinational gate-level circuits by specifying a FSM with a single state and one only transition. (State latch and resetting circuitry are easily removed because they become redundant in the resulting circuit.)

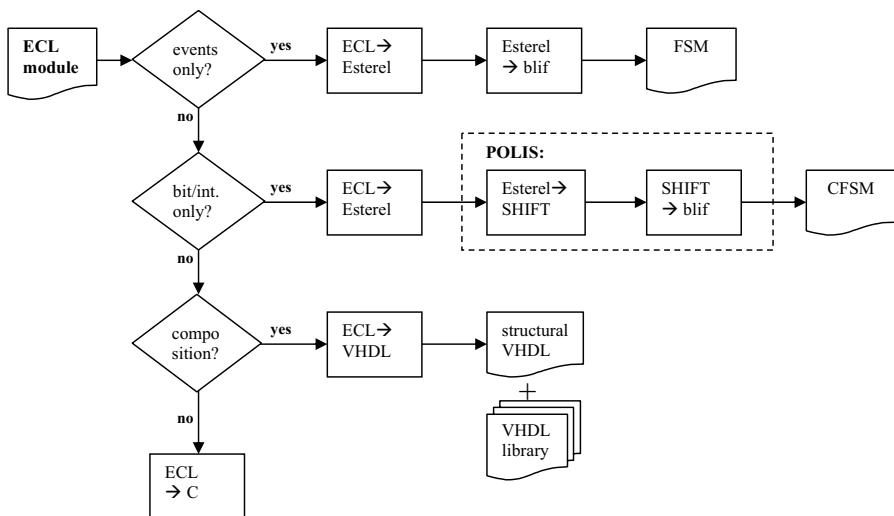


Figure 10.2. Syntheses' paths from ECL.

Similar results can be obtained within POLIS. However, in this case, FSM can include data operations; i.e. EFSM can be synthesized. Again,

combinational circuits can be obtained by having ECL modules with one state and one transition that might include different outputs depending on inputs. As for the data part, the operations that can be considered are those that are supported by Esterel on Boolean and integer data, and the logic (AND, OR, NOT, LSHIFT, RSHIFT, et cetera) and bit operators that we included in the `ECLaddenda.h` file. Therefore, Boolean and integer data is straightforwardly considered within POLIS. Words (bit arrays) are not distinguished from integers; i.e. they are declared and treated as integers. However, we provide designers with some bit-access operators such as `bit_set`, `bit_clear` and `bit_test` so to make descriptions readable. Besides, they are translated to specific, optimized hardware blocks (the inclusion file has also the appropriate C equivalents for simulation or software synthesis). We have adopted this solution because there is no need to modify ECL, neither to adapt any existent tool. The only drawbacks are that the designer should specify the data word lengths apart from the signal/variable declarations (though a simple filter program can handle that), and that the resulting circuits may contain an extra bunch of redundant gates that must be further removed. This last drawback only affects the size of a temporary circuit that must be swept of unnecessary latches and gates anyway.

The previous approaches are fully automatic but the designer loses control of circuit structure and, for large circuits, it is interesting that he/she can decide the structure of a given circuit. In that case, we have devised a possible structural description level in ECL in which all components of the module are instantiated within a parallel construct. Of course, the fine-grain modules of the hierarchy can be synthesized using one of the previous methods (or manually). In this approach, however, a new tool is required to identify such structural descriptions and to translate them into hardware. To make it easy, we have renamed the parallel construct to `compose` so that a program can easily identify these module compositions. Modules may have a VHDL equivalent so no synthesis is necessary (apart from the one that would take VHDL into a gate-level circuit, of course.)

To generate coordinated hardware and software parts, some interconnecting strategy must be planned, and additional modules made to support such interconnections. For small systems, it is possible to use *ad hoc* solutions, though the use of a generic, prefixed communication policy is highly recommended (e.g. POLIS enables different modules implemented in hardware and/or software to communicate with each other at the cost of restricting system behavior to one that fits in its architecture.)

In the work done, we had mainly explored the hardware synthesis of a system checking both possibilities, directly from Esterel and taking Esterel modules into the POLIS tool suite.

## 4. Case Study: A Simple Processor

To test the previously described co-design environment it is clear that it is necessary to check the hardware synthesis aspect of ECL specifications. As ECL supports easy specification and software and hardware synthesis of reactive modules, we have chosen to specify a simple, general-purpose processor. As a consequence, we shall devise the feasibility of our solution to extend ECL capabilities to synthesize data-handling hardware.

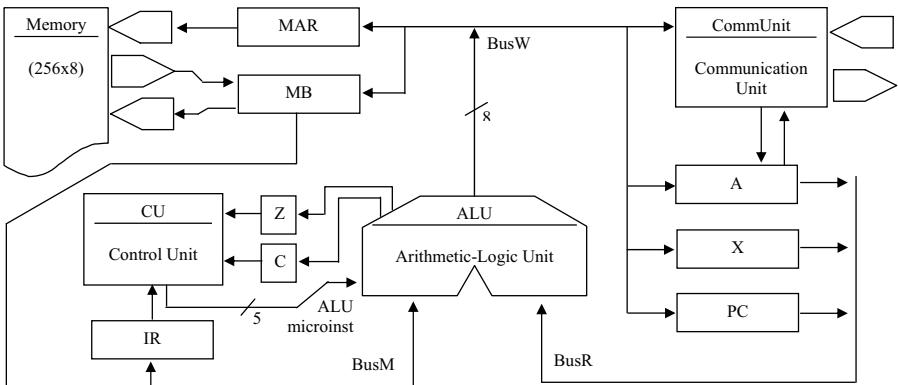
In this section we shall first give an overview of the processor before explaining the structure of its specification in ECL. Finally, we shall present the results of the processor synthesis.

### 4.1 Processor Description

The selected CPU is a simple processor that is used for teaching purposes by the authors. It follows the von Neumann architecture and is made of a processing and a control unit. Obviously, the former is a data-driven system while the latter is a control-driven one. As most of the systems contain subsystems similar to a processor (e.g. algorithmic state machines), this benchmark is proposed to exercise the capabilities in this area of any SLDL, including ECL.

Figure 10.3 shows a detailed scheme of the processing unit (PU), which includes a communicating unit (CommUnit) with request and acknowledgement lines and several I/O ports. As a reduced processor, all registers and buses are 8-bit wide and accompanying memory is a local memory of 256 bytes because it is assumed that communication with the rest of the system to which it is embedded in is done via CommUnit; i.e., I/O ports and ack/req lines. The interaction between the PU and the control unit (CU) is not explicitly shown but for the 5-bit wide line that sends the operation code to the ALU. The CU is, of course, responsible for generating the rest of the controlling orders of buses and registers in the PU.

The instruction set of such processor includes load and store instructions for registers A and X, common arithmetic and logic operations on A, or between A and a memory operand, register-to-register transfers, unconditional and conditional jumps (with relative addressing mode), port input and output, and request sending and acknowledgement waiting operations. To sum it all, the processor has 41 different machine-



*Figure 10.3.* Schematic representation of the processing unit (PU).

level instructions with 5 different addressing modes (immediate, direct, indexed, and indirect for two-operand instructions; and relative for conditional jumps). Execution of these instructions in the PU implies 43 different microinstructions (set of bit-level orders).

## 4.2 ECL Module Structure

Functional specification has not much problem in ECL because of the simplicity of the processor, nor has its simulation in Esterel. The main problem is to refine the specification so to proceed with the synthesis phase, which necessarily is targeted to hardware.

As shown in Fig. 10.4, CU and PU communicate through registers that latch data between clock active edges. For the functional description of such structure we have also designed two modules: CU and PU (for processing unit, but includes the communication unit, too). There is an “integrating” main module that contains the concurrent calls to the rest. It is used to enable Esterel verifying the correct synchronization of the whole and to simulate the processor to validate the functional behavior; i.e. for formal verification and simulation purposes. In this case, we did not plan a module to emulate a memory. Instead, the user must supply appropriate values to memory buffer register (MB) and data\_ready signal (input to CU). Furthermore, he or she must examine the contents of the memory address register (MAR), and of the read and write signals generated by CU. This is a little handicap for simulating a whole program but not for the purposes of this paper regarding the usability of ECL for specifying hardware components.

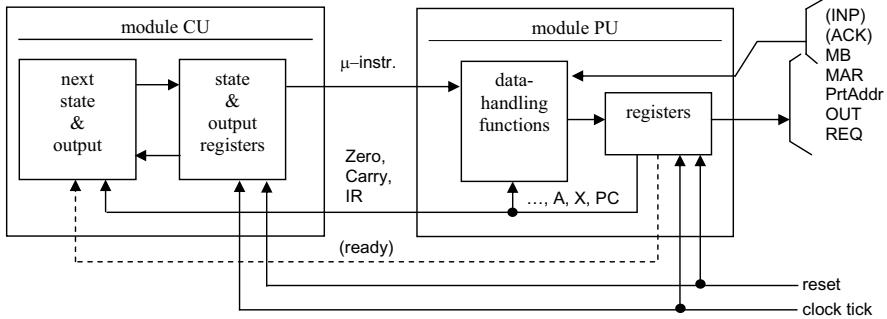


Figure 10.4. Structure of the ECL specification of the simple processor.

CU module corresponds to the reactive part of the processor and is easily described in ECL for both, simulation and synthesis purposes. It emits the necessary micro-orders to the PU and waits for its termination. Usually, PU terminates in one clock cycle and the corresponding data-ready signal would be suppressed. However, to facilitate insertion of pipelining in the PU, it is interesting to keep data-ready signals as a mean for synchronization. Apart from those signals, CU inputs consist of an 8-bit opcode (to select bits of IR, it is necessary to use `bit_test` from `ECLaddenda.h`) and of the zero and carry bits. These inputs are latched at PU output; e.g., the register IR (the container of the current opcode) is synthesized within the PU module.

In fact, all registers in the processor consist of the latches that will be generated to hold data of signals and variables declared in the modules. In the case of IR, as in other registers, we have taken advantage of signal values being available at any time despite its corresponding event to only emitting its contents whenever it changes in the PU module description. (The event is used as data-ready signal.)

PU module had to be described by using operations in the `ECLaddenda.h` file so to ensure the viability of its synthesis. Its behavior is enclosed within synchronization statement bodies and consists of an elementary switch statement in terms of a 6-bit wide microinstruction code.

The description of the whole processor in ECL takes about 400 lines of code whereas the same description in C is about 1000 lines long. Taking into account that the C program is longer because it includes the link to a GUI, the comparison seems to favor a little bit the ECL option. In the following, we shall discuss about the results on synthesizing the ECL description.

### 4.3 Processor Synthesis Results

The synthesis of the processor has been done has been done only partially with only using Esterel, because this approach requires that the module is specified in pure-Esterel. Within this approach, only CU can be synthesized into hardware, once bit selection on IR is replaced by 8 pure signals (i.e. 8 events that are constantly –at every clock tick– fired if corresponding bit is 1.)

Taking that into account, Esterel generated a blif version of about 700 lines that produced a VHDL code about 900 lines long. The resulting circuit has a total of 16 flip-flops for a one-hot state encoding with a glue logic made of approximately 300 gates. Within POLIS, a similar result was obtained, but EFSM states were binary encoded.

The synthesis of PU is only possible within POLIS because it contains data operations. As these operations are expressed in terms of functions included in `ECLaddenda.h`, we had only to create an auxiliary file containing the width of the data before converting the module into SHIFT (an internal language of POLIS). Operators are converted into appropriate subcircuits inside SHIFT, but a renaming (it is done automatically by an accompanying tool) is required so to have POLIS taking them into account for the synthesis. For the PU module, we obtained a 10K-line blif file with 97 latches and about 1250 gates, after having swept redundant gates and latches. However, these results can be further optimized.

We have also tested a structural specification of part of the PU; i.e., describing the module in terms of predefined modules. Predefined modules have a simulation version in ECL and a synthesis version in VHDL. (They accept parameters such as the number of bits.) With this approach, the structural hardware specification in ECL is readily converted into structural VHDL specifications.

Finally, we set up an experiment with a single file containing the whole description of the processor to be synthesized with the help of POLIS. In this case, we had to replace the parallel composition of the main module by a `compose` macro (it does not affect verification by Esterel, nor simulation) so to enable POLIS to import the description and perform the final synthesis steps, which gave similar results that the separate syntheses.

## 5. Conclusion

A good SLDL should enable not only to specify a system at the top-most level of abstraction, but also to describe system components at lower levels of abstraction.

ECL is an extension to the C language with Esterel constructs so to make it easy to specify reactive systems with data-handling parts. Despite this extension, ECL is not intended to be a unique SLDL in the whole design process. Instead, it is addressed to be a language in a language set for specifying large systems devoted to control-dominated components.

In this work, we have devised several forms of describing hardware components with ECL, with emphasis on extending it to data-handling components. While carrying the case study out, we have found that ECL relies on Esterel for interactive simulation of reactive parts and, therefore, the designer must know both languages. Giving this fact, it is possible to complement ECL with some macros to make it easier to implement a variety of preemptive clauses that already exist in Esterel and are very convenient to increase expressiveness. We have developed an additional file to use them as well as to correct a few bugs in the ECL compiler implementation.

As for the hardware synthesis aspects, we tested the ECL capability for describing reactive behavior and synthesizing it into hardware FSM. But the main contribution of this work is about synthesizing the data-handling parts of ECL modules.

The proposed approaches do not modify ECL but only requires the inclusion of a file that contains C macros and functions that enable both simulating the module and a path to hardware synthesis. The simplest of them is to take profit of transition functions of single state FSM to generate simple combinational circuits. The second proposal relies on POLIS to be able to generate both the reactive and the data part of EFSM described in ECL. Finally, the last possibility is to rewrite the behavioral ECL module into some sort of structural hardware description in terms of predefined modules.

The results of these experiments indicate the suitability of ECL for intermediate abstraction levels of specifications in a design framework were a set of SLDL is used. The little weaknesses of ECL expressiveness and flaws of the ECL compiler have been corrected while extending its usage to data-path hardware synthesis. As a consequence, ECL can not only specify reactive, control-dominated systems, but also take into account their data-flow part for both, software and hardware synthesis.

In the short term, we would like to compare results with other languages, especially SystemC. Also, we see very interesting to include ECL as an accompanying tool to other higher-level C-based SLDL, since they could be refined down into ECL for EFSM synthesis. Within this approach, it is possible to take profit of all the advantages of Esterel regarding the verification of synchronous systems.

## References

- [1] F. Balarin, et al. (1997). *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Boston, Mass.: Kluwer Academic Publishers.
- [2] G. Berry, P. Couronné, and G. Gonthier. (1991). “The Synchronous Approach to Reactive and Real-Time Systems”. *IEEE Proceedings*. Vol. 79.
- [3] G. Berry, and M. Kishinevsky. (2000). “Hardware Esterel Language Extension Proposal”. *Workshop on Synchronous Reactive Languages*.
- [4] Celoxica Ltd. *Handel-C Language Reference Manual v3.1*. [www.celoxica.com](http://www.celoxica.com).
- [5] R. Dömer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual v1.0*. [www.specc.org](http://www.specc.org).
- [6] J. Gerlach, and W. Rosenstiel. (2000). “System Level Design Using the SystemC Modeling Platform”. *Workshop on System Design Automation*.
- [7] T. Grötker, S. Liao, G. Martin, and S. Swan. (2002). *System Design with SystemC*. Boston, Mass.: Kluwer Academic Publishers.
- [8] B.W. Kernighan, and D.M. Ritchie. (1988). *The C Programming Language*. Prentice-Hall.
- [9] IEEE. (1993). *Standard VHDL Language Reference Manual: IEEE Std 1076-1993*. IEEE Pub.
- [10] L. Lavagno, R. Passerone, and E. Sentovich. (2001). *ECL 3.0 Documentation*. Cadence Design Systems.
- [11] L. Lavagno, and E. Sentovich. (1999). “ECL: A Specification Environment for System-Level Design”. *Design and Automation Conference (DAC)*. Los Alamitos, CA: IEEE Computer Society Press.
- [12] Ll. Ribas, and J. Saiz. (2002). “A Study of Specification Experiments with ECL”. *Forum on Design Languages (FDL)*. ECSI.
- [13] B. Stroustrup. (1997). *The C++ Programming Language (3rd edition)*. Reading, MA: Addison Wesley Longman.
- [14] S. Swan. (2001). *An Introduction to System-Level Modeling in SystemC 2.0*. [www.systemc.org](http://www.systemc.org).
- [15] S. Swan et al. (2001). *Functional Specification for SystemC 2.0*. Synopsys, Inc., CoWare, Inc., Frontier Design, Inc. and others.

III

## ANALOG AND MIXED-SIGNAL SYSTEMS

*This page intentionally left blank*

A system on chip (SOC) is defined as a complex integrated circuit that integrates the major functional elements of a complete end-product into a single chip or chipset. One big challenge in today's SOC designs is to manage heterogeneity as they more and more encompass digital, analog, mixed-signal hardware, software components and even non electrical mixed-technology parts such as sensors and actuators (MOEMS: micro opto-electronic mechanical systems). The design of such systems therefore requires powerful modeling, simulation and synthesis means to manage all the tasks consistently and efficiently. As modeling tasks are at the heart of the design process, it is of paramount importance that appropriate modeling languages are used consistently throughout the design process. Hardware description languages such as VHDL or Verilog are today routinely used for top-down digital design process from RTL description to gates and design flows based on these languages are fairly well established. At system level, SystemC is becoming a de facto standard language for supporting more abstract discrete-event models of both hardware and software components.

The situation is however more critical when it comes to handling analog, or continuous-time, non electrical parts, and consequently to consistently manage heterogeneous SOC designs. Analog and mixed-signal extensions to the VHDL and Verilog languages, respectively called VHDL-AMS and Verilog-AMS have been recently developed to address this issue. These extended languages provide additional means to describe and simulate the behavior and the structure of mixed-signal and mixed-technology components. They however do not (yet) provide enough simulation efficiency for validating very large and complex heterogeneous SOC's. Tools like Matlab/Simulink do offer a partial alternative at system level but they do not provide a complete seamless path to hardware circuits and systems. Actually, since SystemC is providing an acceptable solution for designing digital SOC's, it is a logical attempt to augment its capabilities to also support mixed-signal and mixed-technology SOC's.

This part gathers a selection of five papers from the contributions made in the AMS workshop at the FDL'03 conference. The papers discuss various important issues that a mixed-signal/mixed-technology SOC designer may have to manage.

The first paper, "Rules for Analog and Mixed-Signal VHDL-AMS Modeling", from Joachim Haase provides insights into modeling and simulation issues that may arise when developing or using user-defined models. On the one hand, the VHDL-AMS language allows for writing models including arbitrary complex differential algebraic equations, pos-

sibly mixed with concurrent and/or sequential/procedural behavior. On the other hand, this descriptive power has to be balanced with simulation issues as the numerical solution of continuous-time equation systems still require well-known mathematical conditions, such as there must be as many equations as unknowns, to be satisfied. The VHDL-AMS language provides a limited support in this respect and the paper shows that additional efforts have to be done on both the model writer and the tool developer sides. The former must follow appropriate modeling guidelines to avoid the ‘no solution’ or non convergence problems, while the latter must provide more powerful debugging tools to help tracking and fixing mathematical and numerical simulation issues.

The second paper, “A VHDL-AMS Library of Hierarchical Optoelectronic Device Models”, from François Mieyeville et al. addresses the behavioral modeling of non electrical parts and shows the development of high-level models from physical equations. The authors outline a modeling methodology in which models are hierarchically organized in such a way system specifications are logically linked to physical device parameters. The paper also demonstrates the capabilities of the VHDL-AMS language to model and simulate a complete optical link including electronic CMOS components, a VCSEL driver and a transimpedance amplifier, and optoelectronic parts, a VCSEL (Vertical Cavity Surface Emitting Laser) and a PIN photodiode.

The paper “Towards High-Level Analog and Mixed-Signal Synthesis from VHDL-AMS Specifications” from Hua Tang et al. addresses the critical task of synthesizing analog and mixed-signal systems from abstract models. The paper describes an automated analog synthesis methodology using VHDL-AMS behavioral models as inputs. A subset of the language that basically allows signal-flow descriptions with some form restrictions is supported. This is mandatory to ease the mapping to architectural elements. Design and performance constraints are also defined in the model. The analog synthesis process does not rely on previously existing designs and uses several techniques such as symbolic analysis and sensitivity analysis to derive a complete circuit from a library of basic blocks, e.g. opamps, GmC, capacitors and resistors. Behavioral models are augmented with layout parasitics to get more accurate performance models for architecture exploration.

The paper “Reliability Simulation of Electronic Circuits with VHDL-AMS”, from François Marc et al. presents a way to include ageing characteristics in a behavioral model. The main advantage of the approach is to get more accurate predictions on how a device, a component or a system will work in time depending on physical effects such as the temperature. Degradation laws are included in the model and may dy-

namically depend on some evolving quantity. In addition, it is possible to accelerate the ageing process to avoid too long simulation times. The presented approach opens the door to the support of arbitrarily complex degradation laws that may depend on the history (states) of a system.

The fifth and last paper, “Extending SystemC to Analog Modelling and Simulation”, from Giorgio Biagetti et al. presents a way to use SystemC to describe and simulate mixed-signal systems. The approach exploits the open architecture of SystemC and its programming capabilities to develop event-driven models of continuous-time behavior based on circuit macromodels. Analog SystemC modules have therefore a specific architecture that allows their simulation by the SystemC kernel as they were regular discrete-event processes. Of course this approach does have limitations such as forcing the discretization of differential equations with a fixed timestep, although each analog block may use its own timestep depending on its activity. Nevertheless, the paper shows that the approach proved to be useful and efficient in the evaluation of the performances of two mixed-signal system, namely a RF transceiver and a mixed-signal fuzzy controller.

I hope this short introduction will encourage the reader to carefully study the papers. The addressed issues are typical of those presented and discussed in the AMS workshop and are likely to be addressed again in future editions.

Alain Vachoux

*Microelectronic Systems Laboratory  
Swiss Federal Institute of Technology Lausanne, Switzerland  
alain.vachoux@epfl.ch*

*This page intentionally left blank*

# Chapter 11

## RULES FOR ANALOG AND MIXED-SIGNAL VHDL-AMS MODELING

Joachim Haase

*Fraunhofer-Institut Integrierte Schaltungen/Branch Lab Design Automation EAS  
Zeunerstr. 38, D-01069 Dresden, Germany*

[Joachim.Haase@eas.iis.fhg.de](mailto:Joachim.Haase@eas.iis.fhg.de)

**Abstract** Since the standardization in 1999 the hardware description language VHDL-AMS is widely used for the modeling of digital, analog, and mixed-signal systems. On the one hand, the powerful language constructs allow a very flexible creation of user-defined models. On the other hand, it is possible to define simulation problems with correct syntax but without a solution. From a mathematical point of view some reasons of such problems are discussed and rules are established that may help to avoid these difficulties.

**Keywords:** VHDL-AMS, modeling rules, consistency of models

### 1. Introduction

Since some years the hardware description language VHDL-AMS has been successfully applied to behavioral modeling tasks [1], [2], [3], [4]. Digital, analog, and mixed-signal systems can be described using VHDL-AMS. The language is a strict superset of the digital VHDL 1076-1993. The evaluation of the digital part of a model is done by an event-drive simulation algorithm. A differential algebraic system of equations describes the analog part. It is evaluated by the analog solver. The interaction between the digital and analog solver is defined as mixed-signal simulation cycle in the VHDL-AMS standard. The exchange of values of digital signals and analog quantities is supported by special language constructs (e. g. 'RAMP, 'SLEW, 'ABOVE attributes, BREAK statement).

All principles of VHDL retain valid for VHDL-AMS models. In the same manner rules that are established for digital VHDL modeling (see e.g. [5], [6]) can also be used for VHDL-AMS models. First papers about specific consequences for analog and mixed-signal models were published [7], [8]. The standard itself requires that the number of simultaneous statements of a model is equal to the declared unknowns of the network system of equations ([1], section 12.6.6). In this chapter we have a look at the fact that the powerful language constructs also allow to create models with correct syntax but without a solution (see Fig. 11.1).

```
entity bench is end entity bench;
architecture without_solution of bench is
  quantity x, y : real;
begin
  x + y == 1.0;
  x + y == -1.0;
end architecture without_solution;
```

*Figure 11.1.* VHDL-AMS simulation task without solution

The background of some of such problems especially with respect to analog and mixed-signal problems is discussed. The difficulties may result from insufficient consideration of the conditions for the existence of solutions and/or insufficient consideration of the VHDL-AMS standard.

The chapter is structured in the following way. At the beginning some requirements concerning analog modeling tasks are summarized. The mathematical propositions that can be applied to these tasks concerning the solution conditions are introduced. Based on these propositions modeling rules are deduced and discussed by means of examples.

## 2. Simulation problem

### 2.1 Elaboration of the analog part

We only consider the analog part in the case of the transient simulation. An analog solution point has to fulfill *characteristic expressions*. The *structural set* of these equations takes into consideration in the case of conservative systems the Kirchhoff's Laws and in the case of nonconservative systems the connection conditions of signal-flow blocks. The *explicit set* is built up by the *simultaneous statements* that describe for instance the constitutive relations of network branches or the input-output relations of signal-flow models. Furthermore, conditions are given

by the *augmentation set*. They add requirements for the initialization phase or at discontinuities (see Table 11.1). The Venn diagram in Fig-

Table 11.1. Default conditions that define the *augmentation set* (without 'DELAYED')

Use of	Initialization phase ( $t=0$ )	Discontinuity ( $t=ts$ )
$Q1'DOT$	$Q1'DOT = 0$	$Q1(ts-0) = Q1(ts+0)$ $Q1$ continuous
$Q2'INTEG$	$Q2 = 0$	$Q2'INTEG(ts-0) = Q2'INTEG(ts+0)$ $Q2'INTEG$ continuous

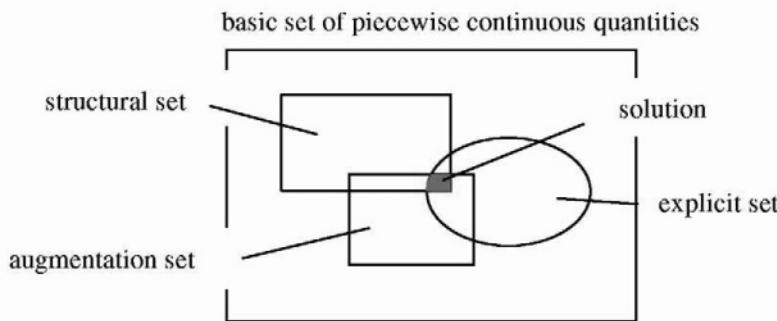


Figure 11.2. Sets that are defined by the analog part

ure 11.2 shows these sets. The diagram demonstrates in a simple way that a solution of the analog part can only exist if conditions that are defined by structural, explicit, and augmentation set are not contradictory. More than one element can belong to the solution set if a characteristic expression does not add a new restriction to the solution but only "duplicates" an existing one. Quantities that have to be determined are across and through quantities of branches (furthermore we use to simplify matters only the terms branch voltages and branch currents) and free quantities.

The VHDL-AMS standard does not define a special system for the analysis of the analog part. For our purposes we use special systems that describe the essential of the problems. Without loss of generality we can assume that the general solutions are characterized in the same

way as for the special systems. The structural and explicit set define a differential algebraic system of equations

$$F(x, \tilde{x}', t) = 0 \text{ with } x : [0, T] \rightarrow \mathbf{R}^n \text{ and } F : \mathbf{R}^{n+q+1} \rightarrow \mathbf{R}^n \quad (11.1)$$

$T$  is the final simulation time.  $q$  is the number of quantities that are used together with the attributes 'DOT' or 'INTEG'. The associated components of  $x$  are summarized in  $\tilde{x}$ . In the initialization phase ( $t = 0$ ) and at discontinuities ( $t = ts$ ) the augmentation set defines

$$B(x(0), \tilde{x}'(0)) = 0 \text{ with } B : \mathbf{R}^{n+q} \rightarrow \mathbf{R}^q \quad (11.2)$$

or

$$B(x(ts), \tilde{x}'(ts)) = 0 \quad (11.3)$$

## 2.2 Characterization of solutions

Existence and uniqueness of solutions can be characterized using

- A conclusion of the inverse function theorem (for details see e.g. [11], p. 52):  $F : D \subset \mathbf{R}^n \rightarrow \mathbf{R}^n$  is a given nonlinear function.  $F'(x^*)$  is regular and  $F'(x^*) = 0$ . The solution  $x^*$  is unique in a neighbourhood of  $x^*$  under certain circumstance.
- A theorem about linear systems of equation (see e. g. [10], p. 260): Assume  $A \in \mathbf{R}^{m \times n}$ ,  $b \in \mathbf{R}^m$ . A solution of  $A \cdot x = b$  exists if  $\text{rank}(A|b) = \text{rank}(A)$ .

These theorems can now be applied to the different simulation domains.

**Initialization phase** In the initialization phase the values  $x(0), \tilde{x}'(0)$  have to fulfill a system of nonlinear equations that is given by (11.1) and (11.2)

$$F(x(0), \tilde{x}'(0), 0) = 0 \quad (11.4)$$

$$B(x(0), \tilde{x}'(0)) = 0 \quad (11.5)$$

Assume that (11.5) can be divided into a linear and a nonlinear part. The linear part consists of  $p$  equations and the nonlinear part consists of  $n + q - p$  equations. Thus, (11.5) has the following structure

$$A \cdot (x(0), \tilde{x}'(0)) - b(0) = 0 \quad (11.6)$$

$$f(x(0), \tilde{x}'(0), 0) = 0 \quad (11.7)$$

with  $A \in \mathbf{R}^{p \times (n+q)}$ ,  $b \in \mathbf{R}^p$  and  $f : \mathbf{R}^n \times \mathbf{R}^q \times \mathbf{R} \rightarrow \mathbf{R}^{n+q-p}$ . Solution conditions can be checked in the following way

- A solution is unique if the Jacobian of (11.5) is regular

$$\text{rank} \begin{pmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial \tilde{x}} \\ \frac{\partial B}{\partial x} & \frac{\partial B}{\partial \tilde{x}} \end{pmatrix} = n + q \quad (11.8)$$

- A necessary condition for the existence of the solution of (11.5) is

$$\text{rank}(A) = \text{rank}(A|b(0)) \quad (11.9)$$

### Remarks

- The solution of the initialization phase is the starting point for the time domain analysis. In the case of a problem (11.1) of higher index it is not sure that the solution of (11.5) is a consistent initial value (see e. g. [12], [13]). These problems can especially occur in non-electrical applications of VHDL-AMS (see also example 3).
- Normally the conditions (11.8) and (11.9) can only be evaluated if an access to the network equations of a simulation engine is possible. But in the case of small test benches the network equations can be directly established in an easy way. This helps to check the application range of the models as we will see in the next section.
- The determination of new initial values after a discontinuity can be done in a similar way. We use argument  $ts$  instead of 0. The function  $B$  has to be replaced w.r.t. table 11.1.
- The ranks of the left and right hand side matrices of (11.9) can be verified for instance as row or column rank of the reduced row echelon forms of the matrices. This can be done for small examples for instance using a program like Mathematica.

**Time domain analysis** After the determination of the operating point in the initialization phase the system (11.1) can be solved in the time domain. This is usually done applying a linear multistep method [7]. A nonlinear system of equation has to be solved at every time step. Furthermore we assume that (11.1) can be divided into  $\bar{p}$  linear and

$n - \bar{p}$  nonlinear equations

$$\bar{A} \cdot (x(t), \tilde{x}'(t)) - b(t) = 0 \quad (11.10)$$

$$\bar{f}(x(t), \tilde{x}'(t), t) = 0 \quad (11.11)$$

with  $\bar{A} \in \mathbf{R}^{\bar{p} \times (n+q)}$ ,  $b(t) \in \mathbf{R}^{\bar{p}}$  and  $\bar{f} : \mathbf{R}^n \times \mathbf{R}^q \times [0, T] \rightarrow \mathbf{R}^{n-\bar{p}}$ . Solution conditions can be checked in the following way

- A necessary condition for the uniqueness of the time domain solution of (11.1) is the regularity of the Jacobian, i. e.

$$\text{rank} \left( \frac{\partial F}{\partial x} + \frac{1}{h} \cdot \frac{\partial F}{\partial \tilde{x}'} \right) = n \quad (11.12)$$

$h$  is the step size.

- A necessary condition for the existence of a solution is

$$\text{rank}(\bar{A}) = \text{rank}(\bar{A}|b(t)) \quad (11.13)$$

*Remark*

- Normally the conditions (11.12) and (11.13) can only be evaluated if an access to the network equations of a simulation engine is possible. But small test bench problems can be directly evaluated.

### 3. Modeling rules

#### 3.1 General rules

The solution of the equations that describe the analog part is given as the intersection of structural set, explicit set, and augmentation set of characteristic expressions. The requirements that are defined by these sets must not be contradictory. On the other hand the solution should be unique. This means for instance that undetermined currents and voltages should be avoided. The following general rules should be taken into consideration

- Each conservative terminal should be connected by some branches to the associated reference node. Otherwise the “node voltage” can be undetermined. *Remark:* This condition is in accordance with the requirement that at least one node 0 must appear in a Spice netlist.
- The branch current of an independent or controlled voltages source is not restricted by a constitutive relation. That is why meshes of

voltage sources should be avoided. Kirchhoff's Voltage Law can easily be violated in such cases. *Remark:* The determination of the reduced row echelon form of the Jacobian (11.12) makes at least one zero row. At least one branch current of a corresponding voltage source is “free”.

- Cuts of current sources should be avoided by the same reasons. In general (11.13) is not valid.
- Kirchhoff's Laws must not be expressed by *simultaneous statements* again. *Remark:* Otherwise some equations of (11.1) are duplicated. The Jacobian (11.12) is no longer regular. A unique solution does not exist.

### 3.2 Initialization phase

Structural set, explicit set, and augmentation set for the initialization phase of the characteristic expressions must not be contradictory. To check this the rules from the previous section can be applied.

A simple test bench for the design entity *integrator(a1)* consists of the model and a waveform source *inp* that is connected with *ain*. The essential equations that describe the test bench are

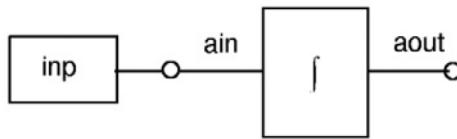
$$\begin{pmatrix} 1 & . & -1 \\ . & . & 1 \\ 1 & . & . \end{pmatrix} \cdot \begin{pmatrix} aout'(0) \\ aout(0) \\ ain(0) \end{pmatrix} = \begin{pmatrix} 0 \\ inp(0) \\ 0 \end{pmatrix}$$

We get

$$\text{rank} \begin{pmatrix} 1 & . & -1 \\ . & . & 1 \\ 1 & . & . \end{pmatrix} = \text{rank}(A) = 2 \quad \text{and (11.14)}$$

$$\text{rank} \begin{pmatrix} 1 & . & -1 & . \\ . & . & 1 & inp(0) \\ 1 & . & . & . \end{pmatrix} = \text{rank}(A|b) = 3 \quad (11.15)$$

for  $inp(0) \neq 0$ . For  $inp(0) = 0$  it follows  $\text{rank}(A|b) = 2$ . That means the model can only be used in cases where  $inp(0) = 0$  and in these cases the solution is not unique (remember (11.9)). The reason is the augmentation set that requires the derivative `ahelp'dot` to be zero in the initialization phase. This can only be fulfilled if the input *ain* is connected to a source with value zero at time 0. On the other hand, *aout* is undetermined at time 0. There is a zero column associated corresponding to *aout* in *A*. The problem can be avoided by replacing the

*Example 1*

$$a_{in} = \frac{1}{ki} \cdot \frac{da_{out}}{dt}$$

```

entity integrator is
  generic (ki : real := 1.0;
           ic : real := real'low);
  port   (quantity ain : in real;
          quantity aout : out real);
end entity integrator;
  
```

```

architecture a1 of integrator is
  quantity ahelp : real;
begin
  ahelp == aout;
  ain == 1.0/ki*ahelp'dot;
end architecture a1;
  
```

*Figure 11.3.* Integrator and wrong VHDL-AMS architecture

default condition for `ahelp'dot` by a condition for `aout`. This can be done for instance by including a `BREAK` statement into the architectural body. The following simultaneous statement equals e. g. `ahelp` during quiescent domain analysis to zero: `break ahelp => 0.0;`

*Example 2*

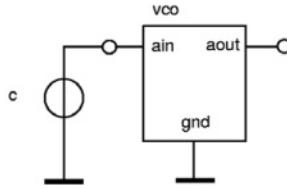
The essential equations that describe the test bench from Fig. 11.4 in the initialization phase are

$$\varphi'(0) - 2\pi \cdot (f_0 + kv \cdot (v_{in}(0) - v_{-f0})) = 0 \quad (11.16)$$

$$v_{out}(0) - Vdc - a \cdot \sin(\varphi(0)) = 0 \quad (11.17)$$

$$v_{in}(0) - c(0) = 0 \quad (11.18)$$

$$\varphi'(0) = 0 \quad (11.19)$$



```

architecture wrong of vco is
  quantity vin across ain to gnd;
  quantity vout across iout through aout to gnd;
  quantity phi : real;
begin
  phi'dot == math_2_pi*(f0+kv*(vin-v_f0));
  vout    == Vdc + a*sin(phi);
end architecture wrong;

```

Figure 11.4. Integrator and VHDL-AMS model that makes the problem

Equation (11.19) results from the default conditions of the augmentation set. The equations (11.16), (11.18) and (11.19) build up a linear subsystem. This can be written in the form

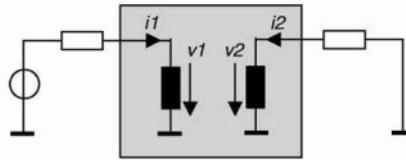
$$\begin{pmatrix} 1 & -2\pi \cdot kv \\ . & 1 \\ 1 & . \end{pmatrix} \begin{pmatrix} \varphi'(0) \\ vin(0) \end{pmatrix} = A \cdot X = \begin{pmatrix} 2\pi \cdot (f0 - kv \cdot v_f0) \\ c(0) \\ . \end{pmatrix} = b$$

$\text{rank}(A) = 2$  and  $\text{rank}(A|b) = 3$  for  $c(0) \neq v_f0 - \frac{f0}{kv}$ . That means the model only works for an input voltage with a special value at time 0 (remember (11.9)). This problem does not occur if (11.19) is replaced by another condition. Either  $vout(0)$  or  $\varphi(0)$  should be specified. This can be done e. g. by including a BREAK statement into the architectural body to fix  $\phi$  during quiescent domain analysis

```
break phi => 0.0;
```

### Example 3

Fig. 11.5 represents a test bench with an ideal transformer. The default conditions given by Tab. 11.1 are replaced. For every choice of the initial values of the inductance currents  $i_1$  and  $i_2$  at time 0 a solution for the initialization phase can be determined.



$$v_1 = L_1 \cdot \frac{di_1}{dt} + \sqrt{L_1 \cdot L_2} \cdot \frac{di_2}{dt}$$

$$v_2 = \sqrt{L_1 \cdot L_2} \cdot \frac{di_1}{dt} + L_2 \cdot \frac{di_2}{dt}$$

*Figure 11.5.* Integrator and VHDL-AMS model that makes the problem

But not all initial values of the inductance currents assure consistent initial values. Consistent values are only given if the following equation is met

$$i_1(0) = \frac{L_1}{\sqrt{L_1 \cdot L_2}} \cdot \frac{R_2}{R_1} \cdot i_2(0) + \frac{E(0)}{R_1}$$

This results from a degeneration of the network equation as already shown in [14], [15]. Equivalent conditions have to be fulfilled after a discontinuity of the voltage source  $E$ . From a VHDL-AMS language point of view it should be taken into consideration that the determination of consistent initial values requires in the transformer model information about the rest of the network.

### 3.3 Time domain simulation

A necessary condition for the unique solution of the time domain analysis problem follows from (11.12):

- The Jacobian of the differential-algebraic system of equations after discretization must be of full rank.

In the time domain analysis a resistive network has to be solved. The derivative  $F$  w.r.t.  $x'$  is zero. The network equations are given by

$$v_{in}(t) - v_1 = 0 \quad (11.20)$$

$$v_r(t) - (r_0 + sl \cdot v_{in}(t)) \cdot i_r(t) = 0 \quad (11.21)$$

$$v_t(t) - v_2(t) = 0 \quad (11.22)$$

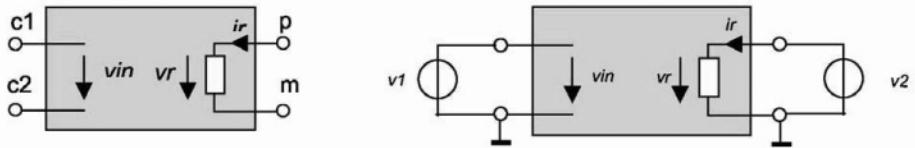
*Example 4*

Figure 11.6. Voltage controlled resistance and test problem with  $vr = (r_0 + sl \cdot v_{in}) \cdot i_r$

The values of  $r_0$  and  $sl$  are constant.  $v_1$  and  $v_2$  are fixed for a given time  $t$ . That means,  $F$  is a function of  $v_{in}$ ,  $v_r$  and  $i_r$ . It follows Jacobian

$$F' = \begin{pmatrix} 1 & . & . \\ -sl & 1 & -(r_0 + sl \cdot v_{in}(t)) \\ . & . & . \end{pmatrix}.$$

The Jacobian is not regular if  $r_0 + sl \cdot v_{in}(t) = 0$ . Some problems can be expected in this case. A unique solution does not exist (see (11.12) and associated explanation). But it is also not clear whether a solution exists at all. Indeed for  $v_{in}(t) = -\frac{r_0}{sl}$  the controlled resistor degenerates to a short branch. An unlimited number of test bench solutions only exists in this case if  $v_2 = 0$ . That means VHDL-AMS must handle  $v_{in}(t) = -\frac{r_0}{sl}$  with special care.

### 3.4 Rules for mixed-signal models

Furthermore some specific problems are the result from the VHDL-AMS standard [1]. The specifics affect for instance the initialization of the real-valued signals that are read by the analog part. If such signals of a real subtype T are not explicitly initialized they are initialized with T'LEFT. Problems during the first call of the analog solver result from this initialization (see e.g. [8]). Another consequence results from the analog-digital simulation cycle. New values can only be assigned to signals at event time points. If a signal assignment is not sensitive to an event the signal will never change its value (see the following example). In some cases this is an unexpected behavior. Therefore the following rules should be taken into consideration

- An explicit initialization of real-valued signals that are used in simultaneous statements is necessary.
- Signals can only be updated if the signal is sensitive to an event.

*Example 5*

In the following, only in architecture *goes* the signal  $vs$  is updated.

```

library ieee; use ieee.math_real.all;
entity bench is end entity bench;

architecture does_not_go of bench is
    signal vs : real := 0.0;
    quantity vq : real;
begin
    vq == 5.0*sin(math_2_pi*1.0E3*now);
    vs <= vq;
end architecture does_not_go;

architecture goes of bench is
    signal vs : real := 0.0;
    signal clk : boolean;
    quantity vq : real;
begin
    vq == 5.0*sin(math_2_pi*1.0E3*now);
    clk <= not clk after 100 us;
    vs <= vq when clk'event;
end architecture goes;

```

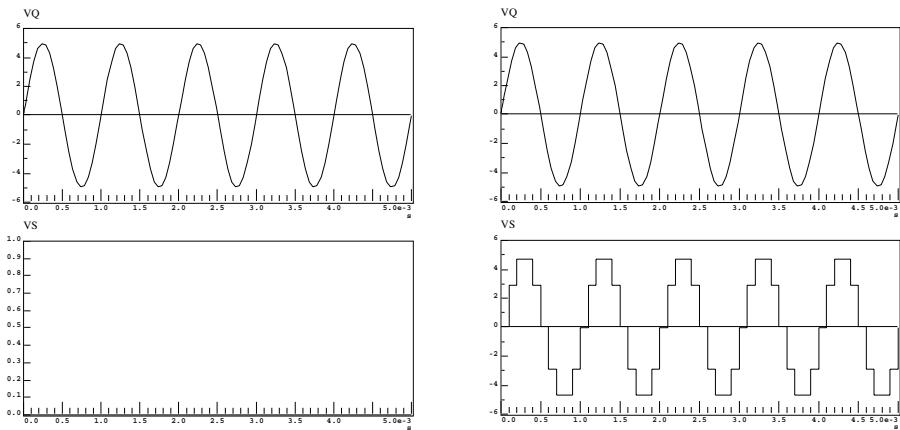


Figure 11.7. Simulation results without and with signal update

The example demonstrates that an assignment of a quantity to a signal only occurs at event time points. A signal assignment that is not sensitive to a digital event does not change its value (left part of Fig. 11.7). The change of the value of a quantity does not produce a digital event. Otherwise, the quantity values at event time points are used for signal assignments (right Fig. 11.7).

## 4. Conclusion

Some rules that should be taken into consideration are discussed in this chapter. They especially consider the solution conditions for the analog part. In addition to the known procedure from Spice-like network simulation engines in the initialization phase and at discontinuities the requirements of characteristic expressions of the augmentation set must be considered. Based on some mathematical proposition some very general rules were given in the section "Modeling rules". A formal verification of the other rules that are founded in section "Simulation problem" demands an access to the network equations. If a commercial simulation engine is used this would not be possible at the moment. The way out is to create small test benches and check the application area of models using these test benches and propositions about their solution. On this base models can be changed and/or special application conditions can be handled with special care. Examples for this procedure are given by several examples. In the same way mixed signal models can be checked. This method helps to avoid problems in VHDL-AMS models and makes the user sensitive to the problems that can occur. A VHPI-AMS Subcommittee of the IEEE 1076.1 Working Group started its activities in 2001 (see e.g. [16]). In the future it can be expected that a VHDL-AMS PLI (programming language interface) will allow to access internal network equations so that the presented methods can be applied. It could be an interesting approach in the future.

## References

- [1] IEEE Standard VHDL Analog and Mixed-Signal Extensions. IEEE Std 1076.1-1999. Design Automation Standards Committee of the IEEE Computer Society, March 18, 1999.
- [2] Christen, E.; Bakalar, K.: VHDL-AMS – A Hardware Description Language for Analog and Mixed-Signal Applications. IEEE Transactions on CAS-II 46(1999)10, 1263-1272.
- [3] Hervé, Y.: VHDL-AMS Application et enjeux industriels. Paris: Dunod, 2002.

- [4] Ashenden, P. J.; Peterson, G. D.; Teegarden, D. A.: The System Designer's Guide to VHDL-AMS. Morgan Kaufmann Publishers, 2002.
- [5] Keating, M.; Bricaud, P.: Reuse Methodology Manual for System-on-a-Chip-Designs. Boston: Kluwer Academic Publishers, 2002.
- [6] VHDL Style Guidelines for Performance by Model Technology Product Support. Last modified: Oct 01, 1999.
- [7] Christen, E.: Selected Topics in Mixed-Signal Simulation. Proc. FDL '02, September 24-27, 2002, Marseille.
- [8] Ruan, K. G.: Initialization of Mixed-Signal Systems in VHDL-AMS. Proc. BMAS 2001, October 10-12, 2001, Fountain Grove Inn Santa Rosa, CA.
- [9] Mades, J.; Glesner, M.: Regularization of hierarchical VHDL-AMS models using bipartite graphs. Proc. DAC 2002, June 10-14, 2002, New Orleans, LA, 548-551.
- [10] Meyberg, K.; Vachenauer, P.: Höhere Mathematik 1. Berlin-Heidelberg-New York: Springer-Verlag, 1990.
- [11] Schwetlick, H.: Numerische Lösung nichtlinearer Gleichungen. Berlin: Deutscher Verlag der Wissenschaften, 1979.
- [12] Pantelides, C.: The Consistent Initialization of Differential-Algebraic Systems. SIAM J. Sci. Stat. Comp. 9(1988)2, 213-231.
- [13] Leimkuhler, B.; Petzold, L. R.; Gear, C. W.: Approximation Methods for the Consistent Initialization of Differential-Algebraic Equations. SIAM J. Numer. Anal. 28(1991)1, 205-226.
- [14] Berg, L.: Einführung in die Operatorenrechnung. Berlin: Deutscher Verlag der Wissenschaften, 1962.
- [15] Reissig, G.; Boche, H.; Barton, P.I.: On Inconsistent Initial Conditions for Linear Time-Invariant Differential-Algebraic Equations. IEEE Transactions on CAS-I 49(2002)11, 1646-1648.
- [16] VHPI-AMS Subcommittee of the IEEE 1076.1 Working Group (documents). Available: <http://www.vhdl.org/analog>

## Chapter 12

# A VHDL-AMS LIBRARY OF HIERARCHICAL OPTOELECTRONIC DEVICE MODELS

F. Mieyeville<sup>1</sup>, M. Brière<sup>1</sup>, I. O'Connor<sup>1</sup>, F. Gaffiot<sup>1</sup>, G. Jacquemod<sup>2</sup>

<sup>1</sup>*LEOM – UMR CNRS 5512, Centrale Lyon, BP 163, 69134 Ecully Cedex, France*

<sup>2</sup>*LEAT - UMR CNRS 6071, 250 rue Albert Einstein, 06560 Valbonne, France*

**Abstract** Internet success and ever-improving microprocessor performance have brought a need for new short range optical communications. The challenge is to integrate electronics, optoelectronic components and optical components on the same chip. Such assembly creates new constraints due to the interactions between different aspects (electronic, optical, thermal, mechanical) that designers have to deal with. Although specific tools have been used to design each module separately, there is no multi-domain simulator or design framework that can meet with such constraints. This paper describes how we can use VHDL-AMS to create a library of hierarchical models and to simulate optoelectronic devices and systems. These models can then be used to propagate specifications from the system down to the physical layer in a top-down approach and to predict the influence of physical parameters on the global performance in a bottom-up approach.

**Keywords:** Optical devices, optoelectronic devices, hierarchical library, hardware description language, behavioral modelling.

## Introduction

Global throughput requirements continue to grow (up to 1Tb/s) [1]. In contrast, electrical interconnections of on-chip and chip-to-chip systems are approaching their fundamental limits and represent a major performance bottleneck [2, 3]. New approaches for systems and architectures will be required. Among all the potential solutions, interconnect currently represents the technology thrust with the largest poten-

tial technology gap. The issues of electromagnetic crosstalk, capacitive loading and signal distortion can be alleviated by the use of optical interconnects.

## **1. From WAN to SAN**

From an historical perspective [4], optical networks are becoming increasingly compact. WANs (Wide Area Networks) and MANs (Metropolitan Area Networks) are now widespread and constitute the backbone of telecommunication networks. The efficiency of optics and the decreasing cost of optical components has led to the development of optical LANs (Local Area Networks) [5]. More recently, SANs (System or Storage Area Networks) use fiber optics to interconnect workstations and/or mass memory units. In the near future, optical interconnects will be introduced inside machines to transfer data between boards or even between IC chips. Active research is underway to study the optical alternative in order to overcome the limitations of metallic interconnects at the chip-to-chip and on-chip level.

As the scale of the network decreases, the specifications of the system change and the designer has to manage a larger number of trade-offs to optimize the performance of the link. For long-haul links (longer than a few decameters), the major requirements deal with data rate and fiber losses. In the case of short range interconnects, optics has mainly to address global throughput, latency, spatial interconnect density, power consumption, technological compatibility and low cost constraints to offer an alternative to metallic interconnections.

## **2. CAD tools for optoelectronic systems**

From an industrial point of view, the development of alternative solutions to current methods cannot exist without computer-aided-design (CAD) tools. Creating such tools makes it necessary to ensure the compatibility of optronic and electronic simulation engines.

The main feature of effective CAD tools for such applications is their multi-domain and multi-designer nature. The wide spectrum of potential applications and users indicates a necessity for the development of tools capable of providing design capabilities at different levels of abstraction (from systems to devices), i.e. to handle multi-domain analysis and to organize links between specific tools in order to enable top-down and bottom-up design.

## 2.1 Behavioral modelling

In the electronic systems domain, global design through the use of standard hardware description languages from the system specification phase down to final circuit design is being established. The hierarchical character of the language provides a natural way of reducing the gap between system and circuit design. Moreover, it can be used even in a larger system design context, such as that of hardware-software co-design or behavioral modelling and simulation of digital signals and systems.

Because of the diversity of component behavior involved in optronic systems, a unique hierarchy, where each level is associated with a unique simulation engine, is not sufficient. An effective CAD tool should bring together different modelling techniques and simulation algorithms. A standard language must then link these tools.

Even if VHDL-AMS is essentially dedicated to electronic system design, non-electronic components and systems can also be described and integrated for use with a single electronic (spice-like) simulator [6]. VHDL-AMS may be considered a candidate able to fulfill the requirements described previously [7].

However, the optical domain presents specific aspects : whereas optical phenomena are driven by Maxwell's equations, the VHDL-AMS standard does not allow the description of partial equations : the modelling of optoelectronic devices requires methods to describe tridimensional temporal phenomena in VHDL-AMS. There are two ways of establishing a behavioral model. The easiest method consists of establishing fitted polynomial equations that link output to input : this is a relatively generic method that can be used on any device, but the resulting model itself is not generic at all. These fitted equations can be extracted from either characterization results or simulation results performed on specific simulator (finite elements methods such as FDTD [8]). The second method consists of reducing physical equations describing the behavior of the device under consideration by approximations (simplification, averaging over surface, ...) : this method gives rise to a generic model but requires the knowledge of physical equations governing device behavior. Both methods were applied to establish our hierarchical library.

The method of validation of each model depends on the hierarchy level of the model. For each component, we validate the low-level model by comparison with experimental or datasheet results. Then, each higher level model is validated by comparison with results of the validated lower-level model.

### 3. A hierarchical library

Considering the design of optoelectronic communication systems, it is of prime interest to be able to propagate global specifications of a system down to each of its components (electronic, optoelectronic or optical device). The elaboration of hierarchical models should enable the transition from system (high-level model) to physical layer (low-level model) : it will then be possible to translate system performance targets to physical constraints. A summary of the methodology is shown in figure 12.1.

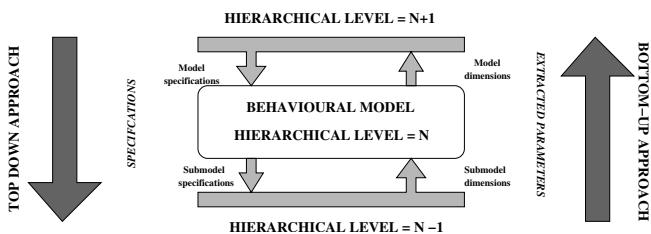


Figure 12.1. Design methodology.

Conversely, the same hierarchical models will enable designers to quantify the influence of physical parameters on global performance and will facilitate the verification phase (bottom-up approach). It is then possible to perform the equivalent of post-layout simulation used in analog design on optoelectronic systems (provided the library is composed of low-level models of each considered optoelectronic device).

If this design methodology requires that models should be based on analytical expressions (which is the case for our models), it can be extended to all types of modelling (implicit equations, tabular methodology) by the use of an adequate design framework [9].

We present a library composed of optoelectronic devices. This library gathers hierarchical models of the following devices : laser, multi-quantum-well (MQW) laser, vertical-cavity surface-emitting laser (VC-SEL), PIN photodiode and BDJ photodiode.

## 4. Optoelectronic devices

### 4.1 Laser and MQW laser

The behavior of a single-mode semiconductor laser can be described by the well-known rate equations [10], [11]. The rate equations describe

the temporal variations of the photon density ( $S$ ) and electron density ( $N$ ). By providing a comprehensive description of physical phenomena inherent to laser behavior, they are well suited to a behavioral description :

$$\frac{dN}{dt} = \eta_i \frac{I}{qV} - \frac{N}{\tau_n} + v_{gr} g'(N)(N - N_0) \frac{S}{1 + \epsilon S} \quad (12.1)$$

$$\frac{dS}{dt} = \Gamma v_{gr} g'(N)(N - N_0) \frac{S}{1 + \epsilon S} - \frac{S}{\tau_p} + \beta N^2 \quad (12.2)$$

$$\frac{d\Phi}{dt} = \frac{1}{2} \alpha \Gamma g'(N) v_{gr} (N - N_{th}) \quad (12.3)$$

The meaning of each parameter and common values are given in table 12.1.

Table 12.1. Parameters of laser devices

Symbol	Parameter	Value
$\tau_n$	Carrier lifetime	$3 \times 10^{-9} s$
$\tau_p$	Photon lifetime	$1 \times 10^{-12} s$
$\Gamma$	Optical confinement factor	0.44
$\epsilon$	Gain compression factor	$3.4 \times 10^{-17} \text{ cm}^3$
$N_0$	Optical transparency density	$1.2 \times 10^{18} \text{ cm}^{-3}$
$v_{gr}$	Lasing medium group velocity	$9.317 \times 10^7 \text{ m.s}^{-1}$
$\alpha_p$	Scattering & absorption loss in cavity	$60 \text{ cm}^{-1}$

The optical output power  $P_{opt}$  of the laser is easy to calculate since it is proportional to the photon density  $S$ . The behavior of a MQW laser differs slightly from the previous one but can also be expressed by the rate equations. The main difference lies in the expression of the optical cavity gain  $G_{laser}$  ( $=g'(N)(N - N_0)$  for a common laser) : the dependency on electronic density becomes logarithmic :  $G_{laser} = g'(N) \ln \left( \frac{N}{N_0} + 1 \right)$ .

The rate equations are the core of the low-level model. Validation of results by comparison with spice-like based models [11], [12] proves the

stability, convergence and precision of the model while being 100 times faster (compared with a Spice-like based model).

A higher level model can be written by expressing all physical parameters in measurable quantities by establishing a new set of parameters (for the high level model) based on the parameters of the low-level model (by combination, aggregation, ...). The development of this “shrinking” method can be found in [13].

## 4.2 Vertical Cavity Surface Emitting Laser : VCSEL

Vertical Cavity Surface Emitting Lasers (VCSELs) are certainly the most attractive light sources to achieve optical communication requirements, not to mention their potentiality in photodetection. Their suitability for monolithic straightforward two-dimensional integration makes it possible to realize dense 2-D arrays and consequently reach high throughput and high spatial compactness. Their technological compatibility with flip-chip techniques and wafer testing contributes to reducing production costs. Their low threshold current and their relatively low operating voltage make them compatible with low power constraints and simple drive CMOS circuits. Moreover, some of their optical properties (single longitudinal mode operation, small divergent output beam, polarization capabilities, ...) make them suitable for either free space or waveguided communication.

The main drawback of VCSELs remains their thermal behavior which strongly alters the threshold current value, the output beam power and polarization. Hence, the design of any optoelectronic system integrating VCSELs requires the use of tools able to take into account not only electrical interactions but also optical and thermal interactions.

**Synopsis of the VCSEL's model.** The behavior of a VCSEL is complex and there are multiple interactions, as shown in figure 12.2. In this figure, “various phenomena” signifies phenomena which exist, irrespective of temperature shift but which are nonetheless affected by it (such as recombinations ...). The “specific phenomena” represents those whose existence is linked to the temperature shift (such as gain spectrum shift...).

Since the numerous interactions cover different domains, it is worth stressing at this point the convenience and relevance of the use of VHDL-AMS.

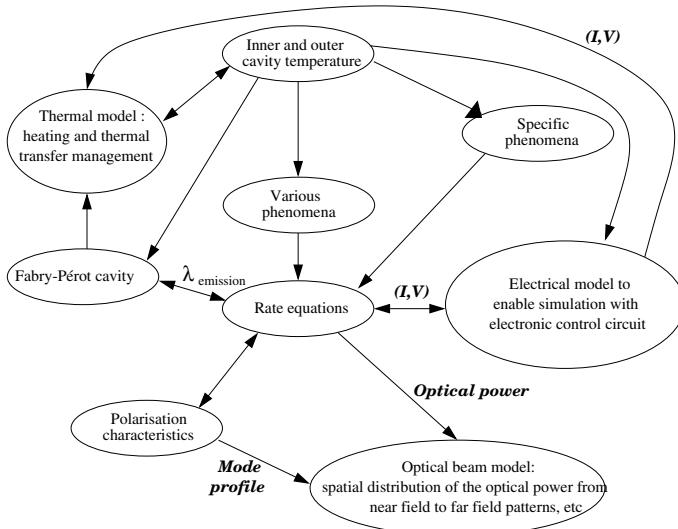


Figure 12.2. Synopsis of the VCSEL.

**The physics-based model.** The use of the well-known rate equations can be extended to the description of VCSEL behavior [14] :

$$\frac{dN}{dt} = \eta_i \frac{J}{qN_{qw}d_{qw}} - v_{gr}g(N, T, \lambda_{FP}) \frac{S}{1 + \epsilon S} - R(N) \quad (12.4)$$

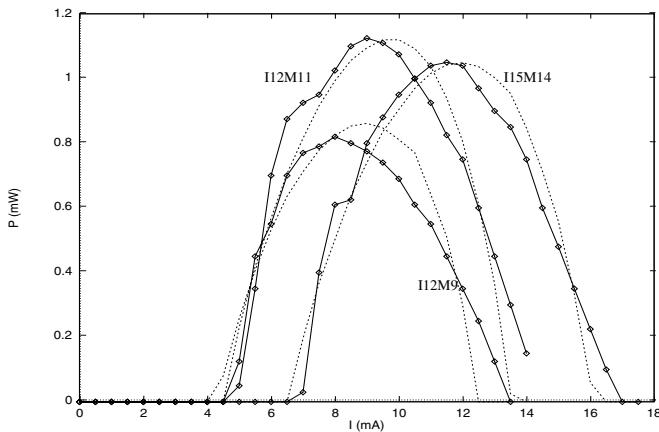
$$\frac{dS}{dt} = \Gamma v_{gr}g(N, T, \lambda_{FP}) \frac{S}{1 + \epsilon S} - \frac{S}{\tau_p} + \beta N^2 \quad (12.5)$$

$$C_{th} \frac{\partial T}{\partial t} = (P_{IV} - P_{optical}) - \frac{\Delta T}{R_{th}} \quad (12.6)$$

This system links the time variation of the carrier density  $N$  and the photon density  $S$  in the cavity, while taking into account  $T$ , the effective cavity temperature, and  $\lambda_{FP}$ , the emission wavelength.  $g(N, T, \lambda_{FP})$  is the analytical expression of the behavior of the optical gain cavity towards input current, voltage and temperature : its formulation ensures the genericity of the model (contrary to commonly used fitted expressions). The third equation (12.6) is a simple monodimensional thermal equation which provides a basic and convenient approximation including heat sources, heat flows and thermal properties of VCSELs [15]. Other parameter meanings can be found in table 12.2.

For the validation of the model, we use top-emitting index-guided Al-GaAs VCSELs processed at LETI (France) [16]. A complete description of the device has been presented in [17].

Fig. 12.3 shows the measured light-current characteristic curves for VCSELs presenting various cavity and aperture diameters. The simulation results are also plotted in this figure.



*Figure 12.3.* Comparison of experimental and simulated (dashed curves) light-current characteristics of VCSELs with various cavity and aperture diameters.

The simulation results fit the experimental curves with a maximum error of 10% ; the results concerning the threshold current match with a maximum error of 5%.

The complete description and validation of the VHDL-AMS model can be found in [17].

This low-level model gave rise to a higher level model whose thermal description is reduced to the calculation of the steady state temperature of the VCSEL working under given conditions. Thus the previous equation system can be rewritten without the equation (12.6). Another higher level model can be used, based on a new set of parameters (resulting from a combination of the physical parameters [13]), consisting of measurable quantities that can be found in any datasheet.

### 4.3 The optical fiber

The fiber used in the optical link (presented further) is a standard monomode optical fiber. In our case, Fourier methods [10], commonly used to model the propagation through optical fibers, cannot be used

Table 12.2. Parameters of VCSEL devices

Symbol	Parameter	Value
$R(N)$	Loss due to recombination	$R(N) = \frac{N}{\tau_n}$
$\tau_n$	Carrier lifetime	$\frac{1}{A + BN + C_{Aug}N^2}$
$A$	QW unimolecular recombination coefficient	$1 \times 10^8 \text{ s}^{-1}$
$B$	QW radiative recombination coefficient	$1 \times 10^{-10} \text{ cm}^3 \text{s}^{-1}$
$C_{Aug}$	QW Auger recombination coefficient	$5 \times 10^{-30} \text{ cm}^6 \text{s}^{-1}$
$\epsilon$	Gain compression factor	$2.5 \times 10^{-17} \text{ cm}^3$
$N_0$	Optical transparency density	$1.76 \times 10^{18} \text{ cm}^{-3}$
$v_{gr}$	Lasing medium group velocity	$9.317 \times 10^7 \text{ m.s}^{-1}$
$\alpha_p$	Scattering & absorption loss in cavity	$60 \text{ cm}^{-1}$
$\lambda_{FP}$	Emission wavelength	863 nm at 300K
$\Delta\lambda_{FP}$	Temperature coefficient of $\lambda_{FP}$	$0.0505 \text{ nm.K}^{-1}$
$\lambda_{p0}$	Gain peak wavelength at 300K	850 nm
$\Delta\lambda_p$	Temperature coefficient of $\lambda_p$	$0.27 \text{ nm.K}^{-1}$
$\Delta\lambda$	FWHM of gain profile	35 nm
$T_{ref}$	Characteristic temperature	240 K

since this requires the knowledge of the entire input signal. We chose to use a parabolic phase digital filter to solve the propagation in monomode optical fiber [18].

As the injected power remains weak, the Kerr effect may be neglected and only linear propagation phenomena are to be taken into account. The output electromagnetic field may be computed as the convolution product of the input electromagnetic field  $E_{opt}(t)$  delivered by any optical source model and the impulse response  $h(t)$  of the fiber :

$$P_{fiber} = A | E_{opt}(t) \otimes h(t) |^2 \quad (12.7)$$

where

$$E_{opt}(t) = \sqrt{P_{opt}(t)} \exp(j\Phi(t)) \quad (12.8)$$

$$h(t) = (1+j)(4\pi\beta_2 L)^{-\frac{1}{2}} \exp\left(\frac{-jt^2}{2\beta_2 L}\right) \quad (12.9)$$

$\Phi(t)$  is the phase of the optical source output beam and  $A, L, D$ , respectively the attenuation, length and global velocity dispersion of the fiber.

The infinite impulse response (IIR) is approximated by a finite impulse response (FIR) numerical filter and implemented with digital process block samples so as to make it possible to simulate the output field in the time domain by a numerical computation of the convolution product.

By using complex coefficients  $a(n)$ , the convolution integral can at last be expressed as follows :

$$P_{fiber}(kT_s) = A \left| \sum_{n=0}^{N-1} a(n) E_{vcSEL}[(k-n)T_s] \right|^2 \quad (12.10)$$

with  $T_s$  the sampling period of the electric field envelope and  $N$  the number of filter coefficient.

To obtain the filter coefficients,  $h(t)$  is truncated and smoothed by the classical time window function  $h_w(t)$  :

$$h_w(t) = 0.54 + 0.46 \cos\left(\pi \left|\frac{t}{NT_s}\right|^3\right) \quad (12.11)$$

A comparison between this time-domain model and a classical BPM algorithm has been carried out for different values of  $N$  (filter order). For  $N = 200$ , the simulation results of the two methods are closer than

5% and computation times are similar (about 1 min for a pseudorandom frame of 64 bits on a Sparc20 workstation).

For more details, the complete model and its validation can be found in [18]. For this component, no higher-level model was developed since the simulation times are compliant with system design constraints.

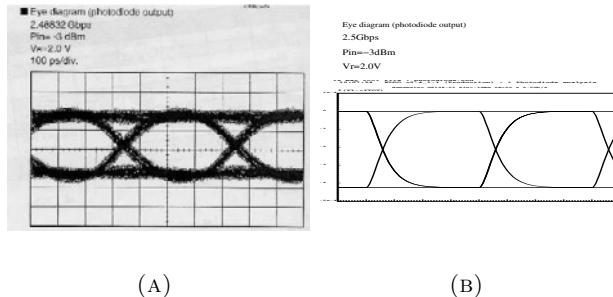


Figure 12.4. Eye diagram ( $P_{IN} = -3$  dBm,  $V_R = 2.0V$ , 2.5 Gb/s) for a PIN Hamamatsu G8198-01 photodiode : (a) datasheet and (b) simulation.

## 4.4 The PIN photodiode

We used a high speed InGaAs PIN photodiode [19] that is well-suited for high-speed optical communications.

**The model.** This model is based on the widely used charge control model [20], which takes into account the photoconversion, the diffusion and the drift of carriers and the junction capacitance.

Two main elements are thus calculated : the junction capacitance which is one of the factors most limiting bandwidth when coupled with interface electronic circuits ; and photoconversion (drift/diffusion time) so as to take into account optoelectronic conversion through the expression of the impulse response of the photodiode [21].

A two-level model has been written : the first one is based on the integration of Boltzmann equations through the PIN and takes into account diffusion and drift effects. The second higher level model extracts, from the datasheet, parameters such as electrical bandwidth, dark current, junction capacitance value at 0V and drift/diffusion time. This high level model was validated by comparison with datasheets. Junction capacitance shows good agreement with a maximal error less than 5% and the eye diagram exhibits the same aperture as shown in fig. 12.4.

**Design methodology illustration.** Our approach (cf. figure 12.1) can be illustrated with the photodiode PIN. The high-level model takes into account responsivity, cutoff frequency, total capacitance and dark current : these parameters can be delivered by the lower level model (*submodel dimensions* in the bottom-up approach) or extracted from datasheet specifications (*model specifications* in the top-down approach). The low-level model takes into account junction saturation current, junction capacitance, photon lifetime diffusion and drift time : these parameters come from the higher level model (*submodel specifications* in the top-down approach) or from the physical structure of the photodiode (bottom-up approach) as shown in figure 12.5.

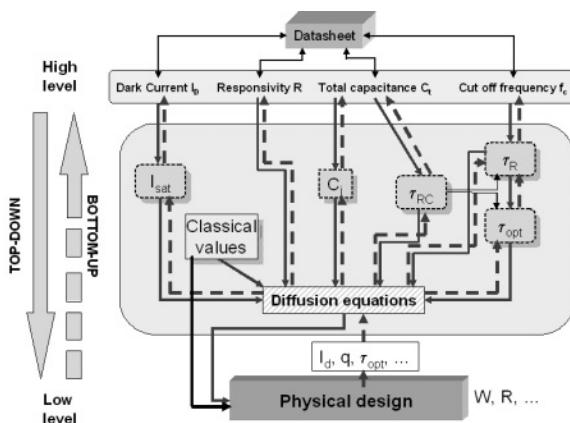


Figure 12.5. Hierarchical aspect of PIN photodiode model

**The BDJ photodiode.** A model of a Buried Double p-n Junction (BDJ) photodiode (used for fluorescence detection and extensively used in biomedical applications [22] : DNA sequencing, clinical chemistry, . . . ) has also been developed.

## 5. An optical link

We used the previously described models to design a complete chip-to-chip optical link processing data at 2.5 Gb/s with a Bit-Error-Rate (BER) equal to  $10^{-18}$  (cf. figure 12.6).

This link is composed of optoelectronic devices and CMOS interface circuits. For the simulation we used VHDL-AMS models for optoelectronic devices and transistor-level descriptions for the VCSEL driver and

transimpedance amplifier (TIA). These interface circuits were designed in a  $0.25\mu\text{m}$  CMOS technology.

## 5.1 Simulation results

Figure 12.7 shows the simulation (performed in a single EDA framework) results of a 2.5Gb/s data exchange. The simulation time is about 2 minutes on a Sparc 10 for a 40-bit frame.

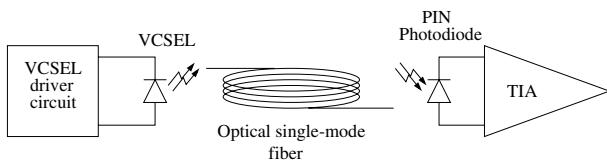


Figure 12.6. Synoptic of the complete link.

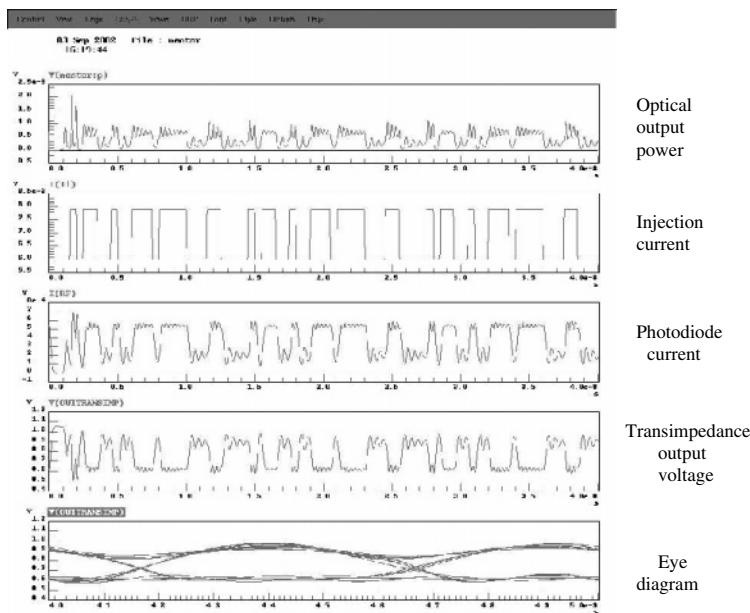
These simulation results can be exploited in many ways at several levels (from system specifications down to technological aspects).

## 5.2 Exploiting results

First, the designers can use these models to verify whether the system meets the specifications and if not to identify the major bottlenecks. It is then possible to optimize the operating conditions of the different subparts of the link : for example, the external efficiency (defined as the ratio of the output optical power to input electrical power) of the VCSEL may be maximized by adjusting the bias current; the BER may also be minimized by modifying the extinction ratio, i.e. the relative values of the bias and modulation current.

Secondly, owing to the hierarchical nature of the VHDL-AMS models, it is possible to go beyond the optimization of operating conditions. Indeed, the physical nature of the model makes it possible to link system specifications to structural and geometrical parameters so as to provide specifications to the device designer, for both electronic components (transistor layout) and optoelectronic devices (sizing of Bragg mirrors for the VCSEL or determination of the intrinsic zone width of the PIN photodiode, ... ) [9].

Thirdly, the high-level models (based on the physical model) allows the designer to predict the performance of the whole link from physical parameters in a verification (or bottom-up) approach.



*Figure 12.7. Simulation results of the complete link.*

## 6. Conclusion and perspectives

This paper presents, for the first time to our knowledge, the possibility of designing an optical link in a standard electronic design framework using the top-down and bottom-up methods commonly used in electronic design. The behavioral modelling approach, using the VHDL-AMS language, allows one to relate system specifications to individual component and subsystem specifications, whatever the nature of the considered component (electronic, optoelectronic or optical). The approach used can permit the design of optical chip-to-chip and board-to-board optical interconnects : simulating both optoelectronic and electronic parts together will allow the quantification of global performance as well as interactions (such as thermal effect). Such simulations also allow the identification of bottlenecks, enabling the designer to distribute design constraints more evenly (according to their intrinsic limitations) over all system components.

## 6.1 Library development

There are at least two levels of model (excepted for the optical fiber) for each optoelectronic component presented in this paper. Each model will undergo the addition of noise models that are currently lacking, and prevent models from being used for exact analysis of optical systems : hence for BER calculation of the previous optical link, only noise from electronic interface circuits was taken into account in a first approach. Furthermore, we are currently developing models for photonic devices (such as waveguides and microring resonators) so as to extend the use of the library and the methodology to on-chip optical systems.

## 6.2 VHDL-AMS limitations

The development of this library allowed some conclusions concerning the limitations of VHDL-AMS. These are of two types. The first constraints come from the language itself : partial derivative equations (PDEs) are not supported in VHDL-AMS specifications. Such a limitation is all the more important for optoelectronic components, since PDEs govern electromagnetic wave propagation. Hence to realize a low-level model, it is necessary to interface VHDL-AMS with a spatial discretization algorithm to the detriment of the unified language description. A second solution, used for our models, consists in reducing the physical description to a monodimensional problem by using surface averaging and approximations.

The second limitation comes from commercial implementations of VHDL-AMS, for which the only numerical resolution algorithms available are those typically found in electrical simulators, such as Newton-Raphson based algorithms, which are not suited to non-linear descriptions. The DC solution is often the main bottleneck : convergence is then achieved by approximations and explicit analytical descriptions so as to alleviate Newton-Raphson limitations.

## 6.3 Methodology conclusion

From a methodology point of view, an extraction of physics-based equations for each model, without any fitted expressions, is possible. The generated models are fully generic and as the high-level models are based on the low-level model, the transitions through the hierarchical levels are facilitated.

## References

- [1] H.J. Zhou, V. Morozov, J. Neff, and A. Fedor, “Analysis of a vertical-cavity surface-emitting laser-based bidirectionnal free-space optical interconnect,” *Appl. Opt.*, vol. 36, no. 17, pp. 3835–3853, June 1997.
- [2] Semiconductor Industry Association : SIA, “International technology roadmap for semiconductors 2000 update,” 2000.
- [3] J.A. Davis et al., “Interconnect limits on gigascale integration (GSI) in the 21st century,” in *Proc. IEEE*, Mar. 2001, vol. 89, pp. 305–324.
- [4] A.F.J. Levi, “Optical interconnects in systems,” in *Proc. of the IEEE*, 2000, vol. 88, pp. 750–757.
- [5] T.H. Szymanski et al., “Terabit optical local area network for multiprocessing systems,” *Appl. Opt.*, vol. 37, no. 2, pp. 264–275, Jan. 1998.
- [6] E. Christen and K. Bakalar, “VHDL-AMS : a hardware description language for analog and mixed-signal applications,” *IEEE Trans. Circuits and Systems II*, vol. 46, no. 10, pp. 1263–1272, Oct. 1999.
- [7] F. Gaffiot, K. Vuorinen, F. Mieyeville, I. O’Connor, and G. Jacquemod, “Behavioral modeling for hierarchical simulation of optronic systems,” *IEEE Trans. Circuits and Systems II*, vol. 46, no. 10, pp. 1316–1322, Oct. 1999.
- [8] P. Bontoux, I. O’Connor, F. Gaffiot, X. Letartre, and G. Jacquemod, “Behavioral modeling and simulation of optical integrated devices,” in *Analog integrated circuits and signal processing*, vol. 29(1), pp. 37–47. Kluwer Academic Publishers, Oct. 2001.
- [9] F. Tissafi-Drissi, I. O’Connor, F. Mieyeville, and F. Gaffiot, “Hierarchical synthesis of high-speed cmos photoreceiver front-ends using a multi-domain behavioral description language,” in *Forum on Specifications & and Design Languages*, September 2003, pp. 151–162.
- [10] G. P. Agrawal, *Fiber-optic communication system*, Wiley Inter-science, 1992.
- [11] P.V. Mena, S.-M. Kang, and T.A. DeTemple, “Rate-equation-based laser models with a single solution regime,” *J. Lightwave Technol.*, vol. 15, no. 4, pp. 717–730, Apr. 1997.
- [12] S.A. Javro and S.-M. Kang, “Transforming Tucker’s linearized laser rate equations to a form that has a single solution regime,” *J. Lightwave Technol.*, vol. 13, no. 9, pp. 1899–1904, Sept. 1995.

- [13] L. Bjerkan, A. Røyset, L. Hafskjær, and D. Myhre, "Measurement of laser parameters for simulation of high-speed fiberoptic systems," *J. Lightwave Technol.*, vol. 14, no. 5, pp. 839–850, May 1996.
- [14] S.F. Yu, "Dynamic behavior of vertical-cavity surface-emitting lasers," *IEEE J. Quantum Electron.*, vol. 32, no. 7, pp. 1168–1179, July 1996.
- [15] S.F. Yu, W.N. Wong, P. Shum, and E.H. Li, "Theoretical analysis of modulation response and second-order harmonic distortion in vertical-cavity surface-emitting lasers," *IEEE J. Quantum Electron.*, vol. 32, no. 12, pp. 2139–2147, Dec. 1996.
- [16] L. Georjon, *Conception et caractérisation de lasers à cavité verticale*, Ph.D. thesis, Commissariat à l'Energie Atomique – Laboratoire d'Electronique de Technologie et d'Instrumentation, Oct. 1997.
- [17] F. Mieyeville, G. Jacquemod, F. Gaffiot, and M. Belleville, "A behavioural opto-electro-thermal vcsel model for simulation of optical links," *Sensors and Actuators A*, vol. 88, no. 3, pp. 209–219, Mar. 2001.
- [18] K. Vuorinen, F. Gaffiot, and G. Jacquemod, "Modeling single-mode lasers and standard single-mode fibers using a hardware description language," *IEEE Photon. Technol. Lett.*, vol. 9, no. 6, pp. 824–826, June 1997.
- [19] "Hamamatsu InGaAs photodiode G8198 series," <http://usa.hamamatsu.com/>.
- [20] G. Massobrio and P. Antognetti, *Semiconductor device modeling using Spice*, McGraw-Hill, 1993.
- [21] J. Graeme, *Photodiode amplifiers : Op Amp solutions*, McGraw-Hill, 1996.
- [22] G.N. Lu, G. Sou, F. Devigny, and G. Guillaud, "Design and testing of a CMOS BDJ detector for integrated micro-analysis systems," *Microelectronics Journal*, vol. 32, pp. 227–234, 2001.

*This page intentionally left blank*

## Chapter 13

# TOWARDS HIGH-LEVEL ANALOG AND MIXED-SIGNAL SYNTHESIS FROM VHDL-AMS SPECIFICATIONS

*A Case Study for a  
Sigma-Delta Analog-Digital Converter*

Hua Tang<sup>1</sup>, Hui Zhang<sup>1</sup>, Alex Doboli<sup>1</sup>

<sup>1</sup>*Department of Electrical and Computer Engineering, Stony Brook University, USA*

**Abstract** This paper presents our experience on high-level synthesis of  $\Sigma - \Delta$  analog to digital converters (ADC) from VHDL-AMS descriptions. The proposed VHDL-AMS subset for synthesis is discussed. The subset has the composition semantics, so that specifications offer enough insight into the system structure for automated architecture generation and optimization. A case study for the synthesis of a fourth order  $\Sigma - \Delta$  ADC is detailed. Compared to similar work, the method is more flexible in tackling new designs, and more tolerant to layout parasitic.

**Keywords:** VHDL-AMS, high-level analog synthesis,  $\Sigma - \Delta$  analog to digital converters, constraint transformation

### 1. Introduction

Currently, there is a severe shortage of efficient high-level synthesis tools for analog and mixed-signal systems. As a result, analog and mixed-signal design continues to be a lengthy and error prone activity, which requires cumbersome design experience and expertise. In a recent paper [14], Gielen and Rutenbar offer an in-depth discussion about present methodologies and tools for analog and mixed-signal synthesis. They conclude that existing work mostly targets circuit sizing and

layout generation, which are low level design activities. Circuit sizing [16], [15], [23] assumes a known circuit topology, and finds the transistor dimensions that optimize circuit performance, such as gain, bandwidth, slew-rate, and power. Layout generation [3], [7], [17], [18] performs transistor placement and wire routing, while contemplating wire parasitic, cross-talk, and substrate coupling. In contrast, analog system synthesis uses high-level descriptions for producing alternate system architectures, and identifying constraints for the architectural components, so that the required system performance is met [9], [11].

High-level analog system synthesis includes four main tasks [9], [11]: (1) specification, (2) architecture (system net-list) generation, (3) performance model generation, and (4) constraint transformation. There is general consensus regarding the need of developing description languages for high-level analog synthesis. These languages should permit specification of a large variety of systems (like filters, converters, PLL, oscillators etc), express both analog and digital functionality and constraints, and provide sufficient insight for automated generation of alternative architectures [12]. In fact, the last requirement implies identifying the “minimum” amount of structural information that has to be present in a specification, so that system architectures result through a systematic process of mapping language constructs to implementation structures.

This paper presents our experience on automated synthesis of  $\Sigma - \Delta$  analog-to-digital converters (ADC) from VHDL-AMS specifications. ADC circuits are critical for many wireless, multimedia and telecommunication applications. The paper summarizes the main characteristics of a proposed VHDL-AMS subset, and then explains its usage for high-level synthesis. VHDL-AMS [1], [6] is a standardized hardware description language that includes constructs for both analog and digital functionality. Considering that extensive knowledge already exists on using VHDL for behavioral digital synthesis, it is explicable to attempt expanding VHDL-AMS to analog synthesis too. The presented VHDL-AMS subset was already used for high-level synthesis of different kinds of analog systems, like signal conditioning applications [8],[9] and filters [11]. This paper shows that functionality descriptions having the *composition semantics* are useful for high-level analog synthesis. For this semantics, the meaning of a system results by composing the meanings of its building elements. Provided that each language construct can be mapped to circuits, correct system architectures are generated by composing the hardware structures for each instruction of the specification [11], [12].

With analog synthesis, the goal is not merely to implement the desired functionality within a given chip area, but also to keep *minimal* the per-

formance degradation due to layout parasitic [7],[17],[18],[21],[22],[23]. Most of the analog system synthesis methodologies follow a top-down design flow, in which layout generation follows system and circuit sizing [2]. However, it is possible that the fixed circuit parameters do not leave enough performance margins to accommodate the layout-induced performance degradations. In this case, the final design would be incorrect. Typical examples include high-frequency filters that incorporate capacitors of the same order of magnitude (tens/hundreds of fF) as the interconnect parasitic [21, 22]. Parasitic capacitors become an integral part of the signal processing performed by the passive elements. Costly re-iterations through circuit sizing and layout generation are needed to produce a constraint satisfying design. The solution is to combine the parameter sizing process with the layout design step to improve design quality and convergence of the CAD algorithms.

The paper presents a constrained transformation method that includes, besides the traditional parameter exploration step, the tasks of block floorplanning, and wire routing. As opposed to other layout-aware synthesis methods [23], the proposed technique is not limited to one application type, as floorplanning and global routing are integrated within the system synthesis method. This allows early contemplation of layout parasitic. Our constraint transformation technique is different from similar work [13] because it includes the additional steps of parameter classification, parameter domain pruning, and identification of parameter dependencies. These tasks improve the convergence of constraint transformation, provided that large parameter domains must be sampled with fine steps. Also, the method is more flexible because it does not require a working design beforehand.

The proposed constraint transformation algorithm uses *symbolic tiles* [19] for compact representation of the floorplan. Though this approach has been used before [7], [17], [19], our representation is much simpler and more compact, thus increases the efficiency of the design algorithms. A smaller number of tiles offers the advantage of faster methods for tile swapping and tile moving. Another difference is that our tiles are *soft* (their sizes and aspect ratio can change). This is a consequence of parameter optimization being part of the synthesis loop. For keeping the complexity of tile managing methods low, complete knowledge about left, right, top and bottom neighbors of each tile must be *explicitly* available. We decided to store the neighborhood information as distinct O trees [20] for each of the four directions. Other representations, such as sequence-pairs [3] or B\* trees [5], are not efficient for our problem as they do not implicitly offer the neighborhood information.

Section 2 discusses the proposed VHDL-AMS subset for synthesis. Section 3 presents the layout-aware synthesis algorithm. Section 4 shows synthesis results for a  $\Sigma - \Delta$  ADC. Section 5 offers conclusions.

## 2. VHDL-AMS Subset for Synthesis

VHDL-AMS [1, 6] is a hardware description language (HDL) for simulation of mixed-domain systems. Currently, it is one of the few standardized mixed-domain HDL.

VHDL-AMS specifications are sets of differential and algebraic equations (DAE) involving continuous-time electrical quantities, like currents and voltages. The system behavior is obtained by numerically solving the DAE sets at time points decided by the simulation cycle of the language [6]. VHDL-AMS specifications do not have the composition semantics, thus they give limited insight into the system structure [12]. Unrestricted VHDL-AMS programs offer poor support for producing architectures. For example, a continuous-time filter specification might include DAE for poles and zeros, phase margin, and quality factor. However, these performance descriptions do not help automatically creating filter architectures through a systematic process of mapping language constructs to circuits [11], [12]. This section presents restrictions that enforce the composition semantics on VHDL-AMS programs.

Signal-flow graphs (SFG) formulate the system behavior as a composition of signal processing operators and signal flow paths. SFG operators are simple linear functions, like addition, subtraction, integration, and multiplication by a constant. Operators generate their outputs depending only on their inputs. There are no influences between connected operators. In our experience [8], [9], [11], [12], SFG provide sufficient insight for automatically producing alternative architectures for a system. To impose the composition semantics on VHDL-AMS programs, we had to identify language restrictions, so that VHDL-AMS constructs can be represented as SFG structures.

The composition semantics requires that connected blocks do not influence each other, so that block outputs depend only on the block inputs. This assumption is widely used for high-level expression of systems, like ADC, PLL, transmitter and receiver systems, to name a few. Enforcing the composition semantics on circuit and transistor level designs is unrealistic. Sections 3 and 4 explain that the analog synthesis flow considers circuit loadings (input/output impedances) and non-idealities (like finite gain and poles) for evaluating the functionality and performance of an implementation. An implicit goal of synthesis is to minimize

the mismatch between the behavior of the implementation and the behavior of the ideal system having the composition semantics.

In [12], we presented a detailed justification of the VHDL-AMS subset for synthesis. VHDL-AMS programs consist of entity declarations, architecture bodies, package declarations, and package bodies. The proposed VHDL-AMS subset for synthesis defines analog signals as free quantities. The subset includes three language constructs: simple simultaneous statements, simultaneous if/case statements, and procedural statements. Following restrictions define the composition semantics for the constructs.

*Simple simultaneous statements* (SSS) express DAE sets. DAE have a broader meaning than SFG, in the general case. We imposed following three restrictions on SSS, so that their semantics can be described as a unique SFG: (1) The left side of an SSS contains only one quantity, or one derivative of a quantity. (2) In a DAE set, a quantity occurs only once in the left side of an SSS. Under the two constraints, the behavior of the left-side quantity is equivalent to the behavior of the output of the SFG structure for the right side of the SSS. (3) The right side of an SSS includes only linear operators such as addition, subtraction, integration, and multiplication by a constant. This restriction is explained by the need to realize these operators using the circuits currently present in our circuit library [8]. The restriction can be loosened, if more circuits are added to the library (for example, mixer circuits).

*Simultaneous if/case statements* (SIS) represent systems with multiple modes of behavior. Digital signals control the activation of the modes. In the context of linear systems, SIS describe variable amplification stages. For example, in [8], the receiver module of the telephone set had a variable gain controlled by a digital signal. We imposed two restrictions for the statements in the SIS branches: (1) All left-side quantities of the SSS in a branch must appear in the other branch, too. Otherwise, the quantity will not have well-defined values for all conditions. Such functionality is incorrect for continuous-time systems. (2) For a quantity present in the left side of two SSS (one SSS for each branch), the operator patterns of the two SSS must be the same. Same operator patterns for two SSS means that their right-side expressions refer to the same quantities, and apply the same operators to the quantities. Different constant values are allowed for the same patterns.

*Procedural statements* (PS) are useful for explicitly specifying SFG structures. PS include instructions like assignments, if/case statements and loop statements. Data dependencies among PS instructions define the signal flow between the SFG blocks for the instructions. As opposed to HDL for digital circuits (like VHDL and Verilog), instruction

sequences do not have a meaning for analog synthesis. The left side of an assignment statement defines the output signal for the SFG corresponding to the right side of the assignment. Assignment statements have the same constraints as SSS, if/case statements must meet the same restrictions as SIS. We discussed in [8] the conditions under which for and loop statements can be translated into semantically equivalent SFG structures. Loop instructions are useful for describing multi-channel systems, like the dual tone multiple frequency decoder (DTMF). DTMF includes a bank of eight filters and amplitude detectors to recognize the presence of a certain tone. The abstract specification of the DTMF uses one for instruction that instantiates the generic filter architecture as part of the loop body. Loop instructions are also useful to express structural regularities of a system. Structural regularity helps setting up compact performance models [10] and simplifying architecture generation [11].

### 3. High-Level Analog Synthesis

Figure 13.1 presents the proposed high-level analog synthesis flow. Rhapsody is the corresponding software environment. Inputs to the flow are VHDL-AMS descriptions of a system, AC and transient domain performance requirements, and physical constraints like area and power consumption. The synthesis output is the sized system architecture (including sized resistors and capacitors, and bounds for op amps/GmC gain, poles, and I/O impedances), the placement of the building blocks, and the global routing of signal and power wires. Given the bounds for the active circuits, op amp/GmC transistors can be sized using state-of-the-art circuit synthesis tools. Following steps form the synthesis flow.

*Architecture generation:* Architectures are netlists of active circuits, i.e. op amps and GmC, and passive elements, such as resistors and capacitors. Two architecture generation algorithms are currently available. The first approach [8, 9] uses a static library of patterns that relate VHDL-AMS language instructions to analog circuits. A pattern matching algorithm produces alternative architectures by composing together the mapped instructions. The second technique [11] is based on the fact that linear signal processing operators can employ either currents or voltages. This observation makes architecture generation similar to a bi-partitioning problem [11]. A set of conversion rules link processing operators and signal types to corresponding active circuits and devices.

*Generation of behavioral performance models:* Symbolic models express the system outputs as functions of inputs and design parameters [10]. The used method generates very compact symbolic expressions by using the structural regularities of a system architecture. The derivatives

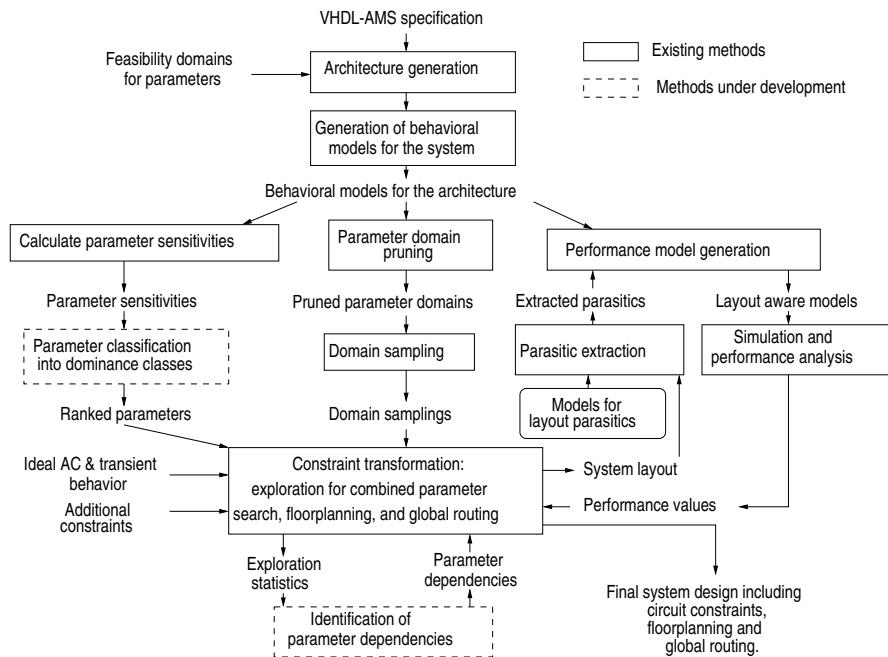


Figure 13.1. Analog synthesis methodology

of state-variables were replaced by their finite differences using integration rules, such as Backward Euler integration. The main reason for behavioral modeling is the long time necessary for SPICE simulations. Section 4 presents several ADC simulations using behavioral models.

*Parameter classification:* Parameters are classified depending on their effect on the system performance. Then, during exploration, dominant parameters are sampled more often. This improves the quality of synthesis because more samples are analyzed for the important parameters. For example, the parameters of the first integrator of a second-order  $\Sigma - \Delta$  ADC have a higher influence on SNR and DR than those of the second integrator [4].

*Parameter domain pruning:* This step eliminates space regions that are unlikely to result in good solutions. In our experience, the convergence of synthesis is poor because design parameters belong to large domains. These large domains must be sampled with fine steps. The synthesis convergence is significantly better, if less promising domains are eliminated. We use interval calculus for domain pruning [21].

*Identification of parameter dependencies:* This step automatically finds dependencies among parameters. For example, the  $gm$  and capacitor values of a filter are related to the system performance attributes, like the 3dB point, quality factor, and resonance angular frequency. The solution

space will include many infeasible points, if parameter dependencies are not considered. This obviously affects the synthesis convergence.

*Parasitic extraction, simulation and analysis:* For each solution visited during exploration, the behavioral models are merged with the extracted layout parasitic to create layout-aware performance models. Models are simulated in the AC and transient domains. After analysis, the obtained attributes guide exploration (as part of the cost function).

*Constraint transformation through combined parameter exploration, block floorplanning and global routing:* Constraint transformation includes (1) finding the block parameters (i.e. circuit gains, poles, input and output impedances), (2) block floorplanning (including finding the aspect ratios and placements of blocks), and (3) wire routing. The combined system parameter exploration and layout generation allows accurate evaluation of layout parasitic. This results in layout awareness of the synthesis methodology [21, 22]. Simulated annealing or tabu search algorithms can be used for the combined exploration step.

The remaining part of this section details the data structures and methods used for system floorplanning.

### 3.1 Tile Representation

A tile [19] based representation is adopted for our layout. Figure 13.2 depicts the tile representation. It represents both active blocks and channels. The active part of the tile is the actual component, and the channel part is the portion of the channels surrounding the active region. The widths of the channel part are denoted by  $\Delta_i$ ,  $i=1,4$ . A layout is a collection of tiles. As compared to the tile definition in [19], the used definition reduces the number of tiles for a layout as it decreases the number of tiles needed to express empty spaces.

As shown in Figure 13.3, tiles can be of three types: (1) active tiles, (2) empty tiles, and (3) margin tiles. The active tile is a tile, which represents an electrical component, such as a resistor, a capacitor, an op-amp etc. A tile, which does not represent an electrical component, is called an empty tile. The tiles at the four borders of the layout are called margin tiles.

A tile is defined by its four *corner-points*. Figure 13.2 shows corner-points as gray bubbles. A corner-point is the pair  $(x, y) \in R^2, 0 < x < w_{max}, 0 < y < h_{max}$ .  $T$  is the set of all tiles. A corner-point belongs to at least one tile. The set of all corner-points is denoted by  $CP$ . A *joint* of a tile is a corner-point, which meets an adjoining tile at a point other than its corner-point. A joint is the pair  $(x, y) \in CP, (x, y) \subseteq T \& \exists t \in T$  for which  $(x, y)$  is not a member but one of its corner-points has its x- or y

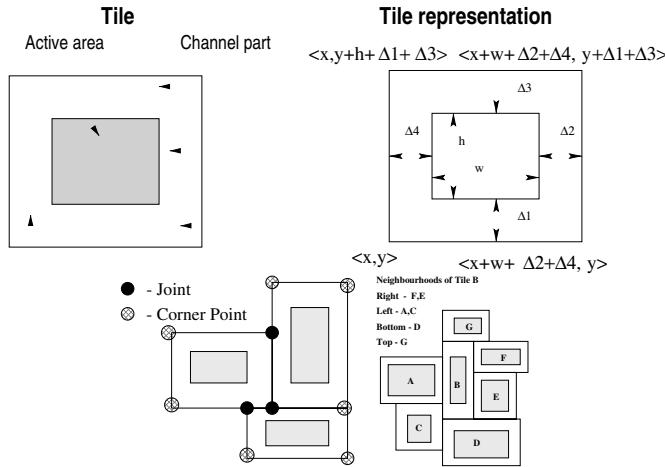


Figure 13.2. Tile definition and tile relations

coordinate equal to  $x$  or  $y$ . The left part of Figure 13.2 illustrates joints as black bubbles.

The relative positions of the neighboring tiles is defined by neighborhood relationships. Four neighborhood relations exist for each tile, as shown in Figure 13.2. The existence of a left neighborhood relation is identified by the constraints (1)  $|y_1, y_1 + h_1 + \Delta_1^1 + \Delta_3^1|$  intersects  $|y_2, y_2 + h_2 + \Delta_1^2 + \Delta_3^2|$  and (2)  $x_1 + w_1 + \Delta_2^1 + \Delta_1^1 + 4 = x$ .  $\Delta_j^i$  corresponds to the channel  $j$  of tile  $i$ . Similarly relationships are defined for the other neighborhoods. The width of the  $\Delta$ -s is determined during the routing phase by the number of nets which are to be routed through the empty spaces of the channel parts. These relationships are stored as distinct O trees [20] for each of the directions. This helps immediate retrieval of the tile neighbors.

### Tile based moves

**Domino Move:** In the domino effect, to accommodate an expansion in a tile in the layout, its neighbors will have to be moved to avoid overlapping. This movement of tiles, called domino effect, moves towards the edges of the layout stopping either when the move is completely absorbed by the empty tiles, or at the border. The domino move function, which implements this phenomenon, is an important function in floorplanning as all other routines will rely on this function.

Figure 13.3 shows an example of a domino move. Tile 7 is to be expanded towards the left. When this tile expands, it overlaps with tiles 4 and 5 which are on its left, as shown in Figure 13.3(c). To avoid the

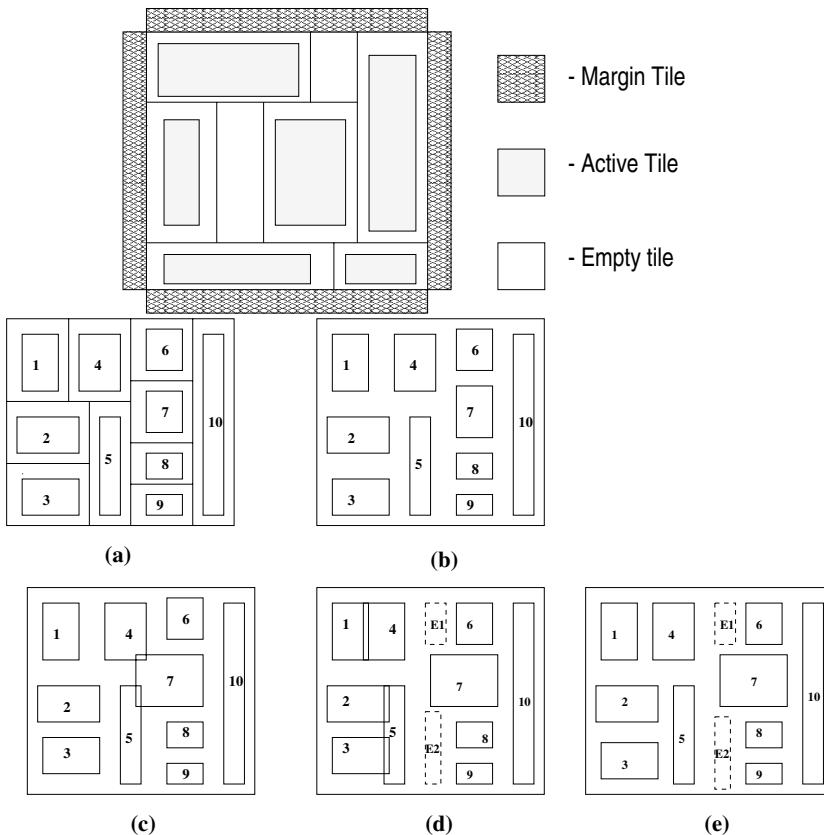


Figure 13.3. Layout and domino move

situation of overlap in the layout, the tiles 4 and 5 are moved to the left by an amount equal to that by which the tile 7 was expanded. These tiles overlap with tiles 1, 2 and 3 (see Figure 13.3(d)). These tiles are moved to the left to avoid overlaps. As the tiles are on the edge of the layout, the chip size is increased so that the tiles are not in overlap with the margin tiles (Figure 13.3(e)). The domino move has a worst case complexity linear with the total number of tiles. Hence, having a reduced number of tiles in the representation helps preserving a low execution time for the domino move step.

**Swap\_tile function:** Given two tiles as an input, this function swaps the two tiles. The dimensions of the tiles are changed to those of each other (the domino move function will be called here). The neighbor lists of the tiles are then interchanged. The coordinates of the tiles are now changed to the coordinates of each other. The swapped tiles are then

resized to their original dimensions. A similar method is used for the `move_tile` step.

**Resize\_tile Function:** This function moves a tile in the desired direction with the specified amount to be moved. This involves a change in the coordinates of the tile. This function is important for searching the circuit parameters of a design, such as the values for resistors and capacitors of a filter.

**Compaction:** As the tiles are moved around and their dimensions varied, there is a danger that the size of the chip may exceed the specified dimensions. To avoid this, compaction of the circuit is done routinely during the iterative design process. The compaction module involves deleting as many unnecessary empty tiles as possible. This procedure would ensure that the circuit dimensions are within limits.

## 4. Case Study

The case study presents a synthesis experiment for a fourth order  $\Sigma - \Delta$  ADC [4]. The goal was to maximize the signal to noise ratio (SNR) and dynamic range (DR) of the ADC.

Figure 13.4 presents the VHDL-AMS specification of the ADC modulator. The architecture body describes the ADC signal flow. It instantiates four times the same stage, as shown by instructions *i1-i4* in the code. This similarity in the specification helped performance model generation, because the symbolic code for a stage was reused four times. Each ADC stage has the same signal processing, but involves different symbolic constants  $g1$ ,  $g1\_prime$ ,  $g2$ ,  $g2\_prime$ ,  $g3$ ,  $g3\_prime$ ,  $g4$ , and  $g4\_prime$ . The numeric values for the symbolic constants were found during constraint transformation. The SFG specification of a stage (starting from statement "*ARCHITECTURE sfg of stage IS*") includes SSS. The quantizer block is described as a process statement sensitive to *ABOVE* events on the continuous quantity  $s5$ .

A rich set of attributes states designer knowledge based constraints. These constraints helped pruning some infeasible solution regions during constraint transformation. According to [4], to keep the SNR loss negligible, the ADC stage bandwidth must be higher than the sampling frequency  $f_s$ , for this example  $f_s = 160\text{kHz}$ . The bandwidth of a signal is defined using tool-specific annotations. Another constraint states that the input signal should be within the range  $[-0.45 \times \text{delta}, 0.45 \times \text{delta}]$ , where  $\text{delta} = \text{VDD-GND}$ . This condition prevents modulator overloading [4]. In our example, the range domain of [-0.2V, 0.2V] for quantity  $s5$  corresponds to this need. The next constraint requires that the gain of a stage is smaller than the gain of the next stage, as the distortions in-

```

-- ATTRIBUTE bandwidth: real;
-- bandwidth = FREQUENCY.((QUANT'voltage -
-- QUANT'voltage(DC) < 3dB) at 1);
ENTITY stage IS
  GENERIC (g1, g2: real);
  PORT (
    QUANTITY vin, vo: IN real;
    QUANTITY o: OUT real);
END ENTITY;
ARCHITECTURE sfg OF stage IS
QUANTITY s: real;
BEGIN
  s == g1 * vin - g2 * vo;
  o == s'integ;
  ASSERT (s'bandwidth > 160kHz)
    REPORT "CONSTRAINT" SEVERITY WARNING;
END ARCHITECTURE;
ENTITY adc IS
  TYPE lim_out IS range GND TO VDD;
  GENERIC (g1,g2,g1_prime,g2_prime,
           g3,g4,g3_prime,g4_prime:real);
  PORT (
    QUANTITY vin: IN real;
    -- IS voltage
    QUANTITY vout: OUT lim_out;
    -- IS voltage)
END ENTITY;
ARCHITECTURE sfg OF adc IS
COMPONENT stage IS
  GENERIC (g1, g2: real);
  PORT (
    QUANTITY vin, vo: IN real;
    QUANTITY o: OUT lim_out);
END COMPONENT;
QUANTITY s3: real range -0.2V TO 0.2V;
QUANTITY s4, s5, s6: real;
SIGNAL c: bit;
VARIABLE delta: real := VDD - GND;
BEGIN
PROCESS (s5'ABOVE(VDD/2)) IS
BEGIN
  IF (s5'ABOVE(VDD/2) = TRUE) THEN
    c <= '1';
  ELSE
    c <= '0';
  END IF;
END PROCESS;
i1: stage GENERIC MAP(g1 => g1, g2 => g1_prime)
  PORT MAP (vin => vin, vo => vout, o => s3);
i2: stage GENERIC MAP(g1 => g2, g2 => g2_prime)
  PORT MAP (vin => s3, vo => vout, o => s4);
i3: stage GENERIC MAP(g1 => g3, g2 => g3_prime)
  PORT MAP (vin => s4, vo => vout, o => s5);
i4: stage GENERIC MAP(g1 => g4, g2 => g4_prime)
  PORT MAP (vin => s6, vo => vout, o => s5);
  IF (c == '1') USE
    vout == VDD;
  ELSE
    vout == GND;
  END USE;
  ASSERT (g1 < g2 AND g2 < g3 AND g3 < g4)
    REPORT "CONSTRAINT" SEVERITY WARNING;
  ASSERT (s5'deriv > 176e3 * delta)
    REPORT "CONSTRAINT" SEVERITY WARNING;
  ASSERT (s3'deriv > 176e3 * delta)
    REPORT "CONSTRAINT" SEVERITY WARNING;
  ASSERT (s4'deriv > 176e3 * delta)
    REPORT "CONSTRAINT" SEVERITY WARNING;
  ASSERT (s6'deriv > 176e3 * delta)
    REPORT "CONSTRAINT" SEVERITY WARNING;
  ASSERT (vin'min>-0.45*delta AND vin'min<0.45*delta)
    REPORT "CONSTRAINT" SEVERITY WARNING;
  ASSERT (vin'max>-0.45*delta AND vin'max<0.45*delta)
    REPORT "CONSTRAINT" SEVERITY WARNING;
END ARCHITECTURE;
CONFIGURATION sd_adc OF adc IS
FOR sfg
  FOR i1: stage USE CONFIGURATION
    WORK.stage;
  END FOR;
  FOR i2: stage USE CONFIGURATION
    WORK.stage;
  END FOR;
  END FOR;
END sd_adc;

```

Figure 13.4. VHDL-AMS specification for a fourth-order  $\Sigma - \Delta$  ADC

troduced by early stages are more important than those of latter stages. The next four constraints (the assert statements involving quantities  $s3'deriv$ ,  $s4'deriv$ ,  $s6'deriv$ , and  $s5'deriv$ ) refer to the slew-rate of the integrators,  $SR > 1.1 \frac{\text{delta}}{T_s}$  [4]. Finally, the limited output swing of the first integrator imposes a range constraint for signal  $s3$ .

Note that the specification does not include any circuit-level details, such as opamp finite gain, finite bandwidth, output swings, and mismatches. These constraints are important for the implementation of the modulator [4]. However, they represent physical details, and thus, should not be present in the high-level specification.

Using the algorithm discussed in [9], four different architectures were contemplated for each ADC stage. Figure 13.5(a) shows the selected implementation. The system-level performance evaluation module identified a sequence of four series configurations for the stages. Symbolic models for series configurations were selected from the library to re-

late the input voltages and currents to the currents and voltages at the quantizer input [10]. The time-domain model for  $Vout(t) = f(Vin)$  was completed by using the symbolic expressions that relate the voltages and currents at the ports of one stage. Based on the small signal models for transconductor and opamp circuits, each stage was modeled as a six terminal block, as shown in Figure 13.5(b). Time-domain symbolic relationships between  $Iin$ ,  $Vin$ ,  $Iout$ ,  $Vout$ ,  $I1$ ,  $V1$ ,  $I2$ ,  $V2$  were calculated by solving Kirchhoff's current and voltage laws, and replacing the derivatives of the voltage drops on the capacitors with finite differences (according to Backward Euler Integration rule).

Constraint transformation and component synthesis explored about 30,000 solution points in three days. For a sin wave signal of 625kHz at the input of the  $\Sigma - \Delta$  modulator, the maximum DR and the output spectrum are plotted at the bottom of Figure 13.5. The maximum SNR is 64db, and DR is about 70db. The design quality is similar to that of the modulator reported in [13]. However, the proposed method is more flexible than that in [13], because it does not assume a working design beforehand. Also, we were successful in synthesizing a higher-order converter, which is more challenging to design. The figure also shows the importance of using detailed circuit models for synthesis, such as models that include poles and zeros rather than ideal models. The two plots with dotted lines correspond to simulations, which used circuit macromodels with one or two poles. In the first case, the circuit still worked as an ADC, but the SNR went down by about 13dB and the DR by about 12dB respectively due to the poles. In the last case, the poles prevented the ADC from a correct functioning.

## 5. Conclusion

This paper presents our experience on high-level synthesis of  $\Sigma - \Delta$  ADC from VHDL-AMS descriptions. We showed that functional descriptions having the composition semantics offer sufficient insight into the system structure for automatically creating alternative architectures. Then, different mappings of VHDL-AMS descriptions to library circuits can be contemplated during synthesis, as the best architecture depends on the targeted performance. The paper also discusses the high-level analog synthesis methodology, and details the layout-aware constraint transformation step. Constraint transformation executes combined parameter exploration, block floorplanning, and global wire routing. The technique was successfully applied to high-level synthesis of a fourth-order continuous-time  $\Sigma - \Delta$  ADC. Compared to similar work, the

methodology is more flexible (due to its capability of accepting VHDL-AMS specifications), and more tolerant in tackling layout parasitic.

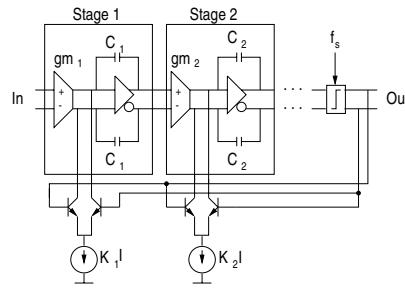
**Acknowledgements.** This work was supported by Defense Advanced Research Projects Agency (DARPA) and managed by the Sensor Directorate of the Air Force Research Laboratory USAF, Wright-Patterson AFB, OH 45433-6543.

## References

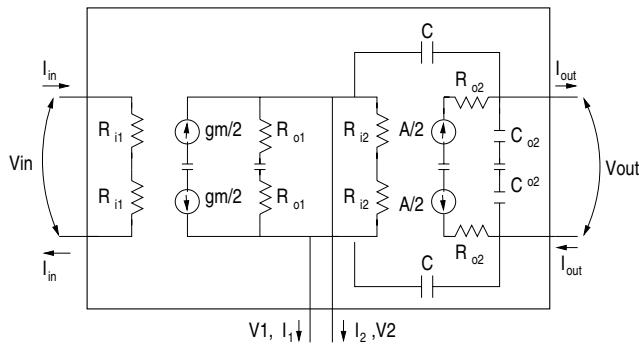
- [1] "IEEE Standard VHDL Language Reference Manual", IEEE Std.1076.1
- [2] H. Chang *et al*, "Top-Down Constraint Driven Methodology for Analog Integrated Circuits", *Kluwer*, 1997.
- [3] F. Balasa *et al*, "Module Placement for Analog Layout Using the Sequence-Pair Representation", *Proc. of Design Automation Conference*, pp. 274-279, 1999.
- [4] J. Cherry, W. M. Snelgrove, "Continuous-Time Delta-Sigma Modulators for High-Speed A/D Conversion", *Kluwer*, 2000.
- [5] Y. Chang *et al*, "B\* Trees: - A New Representation for Non-Slicing Floorplans", *Proc. of Design Automation Conference*, 2000.
- [6] E. Christen, K. Bakalar, "VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications", *IEEE Trans. Circuits & Systems - II*, Vol. 46, No. 10, 1999.
- [7] J. Cohn *et al*, "Analog Device-Level Layout Automation", *Kluwer*, 1994.
- [8] A. Doboli, R. Vemuri, "A VHDL-AMS Compiler and Architecture Generator for Behavioral Synthesis of Analog Systems", *Proc. of DATE*, 1999, pp. 338-345.
- [9] A. Doboli, *et al*, "Behavioral Synthesis of Analog Systems using Two-Layered Design Space Exploration", *Proc. Design Automation Conference*, 1999.
- [10] A. Doboli, R. Vemuri, "A Regularity Based Hierarchical Symbolic Analysis Method for Large scale Analog Networks", *IEEE Trans. Circuits & Systems - II*, Vol. 48, No. 11, 2001.
- [11] A. Doboli, R. Vemuri, "Exploration-Based High-Level Synthesis of Linear Analog Systems Operating at Low/Medium Frequencies", *IEEE Trans. CADICS*, Vol. 22, No. 11, 2003.
- [12] A. Doboli, R. Vemuri, "Behavioral Modeling for High-Level Synthesis of Analog and Mixed-Signal Systems from VHDL-AMS", *IEEE Transactions on CADICS*, Vol. 22, No. 11, 2003.

- [13] K. Franken *et al*, “DAISY: A Simulation-Based High-level Synthesis Tool for Sigma-delta Modulators”, *Proc. ICCAD*, 2002.
- [14] G. Gielen, R. Rutenbar, “Computer Aided Design of Analog and Mixed-signal Integrated Circuits”, *Proc. of IEEE*, Vol. 88, No 12, Dec 2000, pp. 1825-1852.
- [15] M. Hershenson, S. Boyd and T. Lee, “Optimal design of a CMOS op-amp via Geometric Programming”, *IEEE Trans. CADICS*, Vol.20, NO.1, Jan 2001, pp. 1-21.
- [16] M. Krasnicki *et al*, “MAELSTROM: Efficient Simulation-Based Synthesis for Custom Analog Cells”, *Proc. Design Automation Conference*, 1999, pp. 945-950.
- [17] K. Lampaert, G. Gielen, W. Sansen, “Analog Layout Generation for Performance and Manufacturability”, *Kluwer*, 1999.
- [18] E. Malavasi *et al*, “Automation of IC Layout with Analog Constraints”, *IEEE Transactions CAD*, Vol. CAD-15, no. 8, pp. 923-942, August 1996.
- [19] J. Osterhout, “Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools”, *IEEE Trans. on CAD*, Vol. CAD-3, No. 1, pp. 87-100, January 1984.
- [20] Y.Pang *et al*, “Block Placement with Symmetry Constraints based on the O-tree Non Slicing Representation”, *Proc. of Design Automation Conference*, pp. 464-467, 2000.
- [21] H. Tang, H. Zhang, A. Doboli, “Layout-Aware Analog System Synthesis Based on Symbolic Layout Description and Combined Block Parameter Exploration, Placement and Global Routing”, *Annual Symposium on VLSI (ISVLSI)*, 2003.
- [22] H. Tang, A. Doboli, “Employing Layout Templates for Synthesis of Analog Systems”, *Midwest Symposium on Circuits and Systems*, 2002.
- [23] P. Vancorenland *et al*, “A Layout-Aware Synthesis Methodology for RF Circuits”, *Proc. of ICCAD*, 2001, pp. 358-362.

Library component level description of the ADC:



(a)  
op amp/OTA level modeling of an ADC stage



(b)

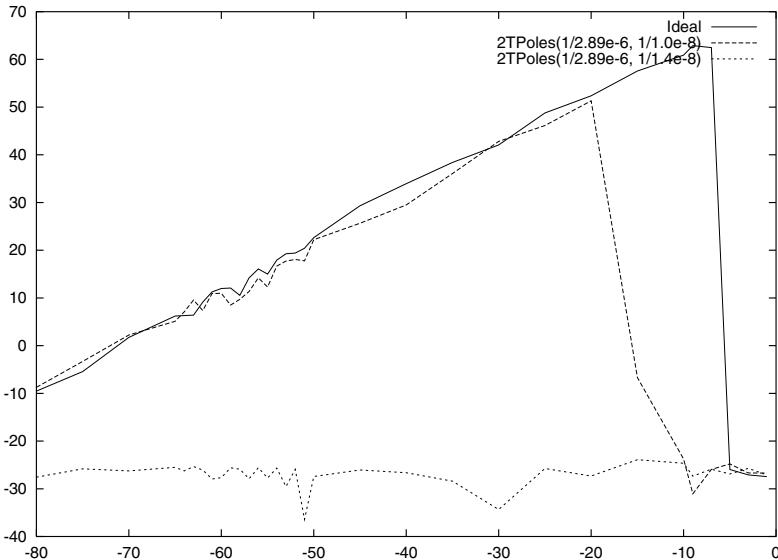


Figure 13.5. ADC architecture, stage model, and SNR and DR plots

## Chapter 14

# RELIABILITY SIMULATION OF ELECTRONIC CIRCUITS WITH VHDL-AMS

François Marc, Benoît Mongellaz, Yves Danto

*Laboratoire IXL, ENSEIRB, Université Bordeaux 1 , UMR CNRS 5818*

*351, cours de la Libération, F-33405 Talence cedex, France*

*marc@ixl.u-bordeaux.fr*

**Abstract** Because of the evolution of the reliability of electronic circuits toward very low failure rate, the statistical analyses through accelerated ageing experiments become too expensive. Today, a deterministic approach of the physics of failure is necessary to estimate the life duration of the circuits. As the ageing mechanisms are highly dependent on the circuit operating conditions, electrical simulation is a very useful tool to help the assessment of the degradation effect. This paper will present on the basis of a practical case the advantages to use a behavioural modelling language like VHDL-AMS for the simulation of ageing of electronic circuits and an original approach for the simulation of the ageing of complex systems.

### 1. Introduction

The life duration of electronic components is usually analysed from a statistical point of view. A set of components, assumed to be identical and to operate in an identical environment will fail at different times following a probabilistic law. This law is usually based on ageing experiment and statistical analyses. Today, this purely statistical approach has reached its limits. Because of the very low failure rate of modern electronic circuits (around  $10^{-8}$  failure per hour and per component), an accelerated ageing experiment needs thousands of circuits to observe only one failure during 2000 hours. Therefore, in many cases, the statistical approach is now excessively expensive. Another approach consists in introducing some determinism in the assessment of reliability. Indeed, the failure of a circuit is the consequence of a physical degradation mech-

anism: the knowledge of the degradation law of this mechanism should make possible the knowledge of the life duration of the circuit. The statistical dispersion of the life duration that we observe practically is the consequence of two causes: the dispersion of the stresses applied to the components due to the different mission profile and the dispersion of the resistances of the circuits to these stresses [1]. For some degradation mechanisms, the degradation laws highly depend on some voltages or currents applied to elementary components. These electrical quantities are usually related to the input signals of the circuit. Therefore, for such mechanisms, electrical simulations can bring useful information to take into account the effect of the application on the circuit ageing and to help to assess the effect of ageing of a component on the application reliability.

This paper will present, with the example of the hot carrier degradation mechanism of MOS transistors on an operational transconductance amplifier, the state of the art of the existing ageing simulation techniques and some original experiments based on behavioural modelling to extend the field of application of ageing simulation to complex circuits and systems. The numerous advantages of behavioural modelling for ageing simulation will be demonstrated.

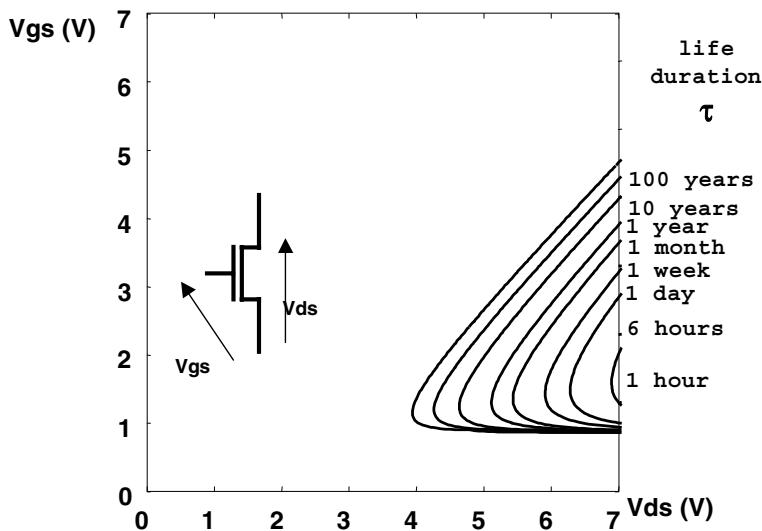
## 2. A degradation mechanism: hot carrier degradations

A degradation mechanism becoming more and more critical with the shrinking of the MOS transistors dimensions is the hot carrier induced degradation. This phenomenon occurs in saturated MOS transistors: the high electric field in the channel accelerates the carriers and injects them in the gate oxide where they are trapped or where they generate interface defects. Consequently, the electrical parameters of the transistor slowly drift, especially the threshold voltage. This phenomenon is not very sensitive to technological dispersions but is highly dependent on drain and gate voltages. A semi-empirical model was proposed by C. Hu [2], linking the life duration  $\tau$  for an arbitrary shift of the threshold voltage to the drain  $I_{ds}$  and bulk  $I_{bulk}$  currents:

$$\tau = \frac{C}{I_{ds}} \left( \frac{I_{bulk}}{I_{ds}} \right)^{-m} \quad (14.1)$$

The parameters C and m represent technological parameters. The figure 14.1 shows the life duration of a transistor using this model, for arbitrary technological parameters and for an arbitrary threshold voltage shift as a function of the gate-source V<sub>gs</sub> and drain-source V<sub>ds</sub> voltages.

This figure demonstrates the high life duration dependency on voltage stresses and the limited domain of critical voltages.



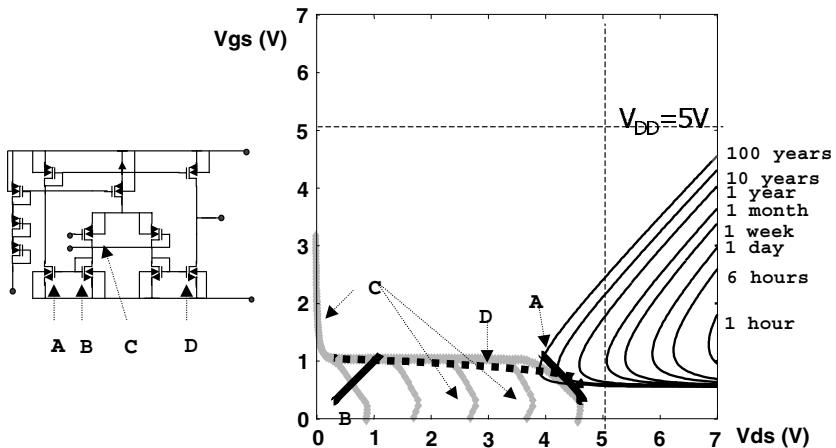
*Figure 14.1.* Life duration of a MOS transistor under hot carrier degradation, as a function of constant voltages at its terminals (Hu model with arbitrary parameters)

In practical situation, each transistor is included in a circuit. This circuit imposes the possible values of the pair  $(V_{ds}, V_{gs})$ . The figure 14.2 shows the possible values of these stress voltages for four transistors of an amplifier, for different values of input voltages. This figure demonstrates that some transistors are more exposed to the hot carrier degradation phenomenon than other and that the intensity of the degradation depends on the input voltages of the circuit. The values of the stress voltages of each transistor have been obtained by static electrical simulation. In real cases, these voltages depend on time and the time duration is a function of all the successive states of voltages. To assess the life duration of the amplifier in a real situation, it is necessary to integrate the effect of degradation during time.

### 3. The reliability simulation today

The reliability simulation, that is most of the time a simulation of the ageing, is the use of simulation to predict the effects of degradation mechanisms on an electronic system. The life duration of an analogue circuit is highly dependent on the application because of the input signals and of the arbitrary criterion used in the definition of the threshold of failure. Therefore, the reliability simulation aims to predict more objec-

tive data like the circuit parameters drift or the evolution of its outputs. The simulation must take into account the physical environment of the circuit (temperature, pressure...), the supply voltages and also the input voltages.



*Figure 14.2.* left : Schematic of an Operational Transconductance Amplifier.  
right : Domain of the accessible ( $V_{ds}$ ,  $V_{gs}$ ) values for 4 transistors of the amplifier for different possible values of the inputs.

The figure demonstrates that the transistor A is critical, B is not critical, C and D are critical for some values of input voltages.

Many softwares like BERT [3] and PRESS [4] have been created since the beginning of the 90's. All are based on the SPICE simulator or some equivalent simulators. They all need the usual data for electrical simulation like the netlist, the electrical models of the elementary components and the operating electrical conditions. For ageing simulation, the ageing laws of elementary components and their parameters values are also needed. This latter information is generally obtained from accelerated ageing and model extractions. Up to now, the most often simulated degradation mechanism in integrated circuits are the hot carriers degradation, the electromigration and the oxide breakdown. The last two mechanisms highly depend on the fine structure of the materials. Then, their simulation gives estimation of failure probability. All ageing simulation tools developed until today are based on the same principle: a SPICE-like electrical simulator manages the electrical simulation while another software uses the simulation results to update the electrical parameters values (using the degradation models) and throws another electrical simulation. Consequently, the ageing simulation is mainly managed out of the electrical simulator. A consequence of the

method is that the simulations are longer than a simple electrical simulation: the most complex circuits that can be simulated cannot contain more than a few tenths of transistors.

#### 4. Behavioural modelling for ageing simulation

A possible solution to speed the simulation of complex circuits is to use behavioural modelling. The behavioural modelling has several other advantages for ageing simulation. The most evident one is that the degradation law of each component, usually known as an empirical equation, is easy to describe with a behavioural language like VHDL-AMS. The second advantage is that the constant parameters of the component (described as GENERIC in VHDL-AMS) can be transformed into variable quantities that can evolve because of the degradation mechanism. These quantities can become inputs of the entity or internal quantities by simply moving them in the PORT declaration or in the ARCHITECTURE. This principle is illustrated in figure 14.3 for a simple MOS transistor model.

Using this method, we can simulate the ageing of a component in a single simulation without external software to manage ageing effects: the component parameters evolve slowly during simulation time because the degradation law is inside the component model. This degradation law can be represented in two ways: either the evolving parameter is described as a function of time, or its first derivative is expressed as a function of electrical quantities. The former expression is applicable only for static electrical condition. The latter is a more general and more realistic expression. For example, the Hu model relation (14.1) for hot carrier degradation should be transformed in a differential equation like:

$$\frac{dV_T}{dt} = BI_{ds} \left( \frac{I_{bulk}}{I_{ds}} \right)^m \quad (14.2)$$

where B depends on the C parameter of (14.1) and of the an arbitrary shift of the threshold voltage chosen as failure condition.

However, this method introduces a difficulty: the order of magnitude of time for the electrical phenomenon is around 1 second or less but the order of magnitude of time for ageing phenomenon is from 1 hour to several years. Most electrical simulator cannot manage so long time values (the simulator we use is limited to approximately 150 minutes of simulated time) and none can manage so different orders of magnitude in a single simulation. Indeed, the simulation of the ageing of a circuit with an input signal around 1 kHz during one simulated year needs more

```

ENTITY nmos_degradable is
  GENERIC (
    W : REAL;
    L : REAL;
    -VT : REAL; — removed
    — initial value of VT
    VT0 : REAL;
    — parameters for hot carrier degradation
    B,M : REAL,
    ageing_time_scale_factor : real=1.0;
    — ...other param.
  );
  PORT ( TERMINAL Tdrain, Tgrille,
         Tsource, Tbulk: ELECTRICAL );
END ENTITY nmos;
ARCHITECTURE mos1 OF nmos_degradable IS
  QUANTITY vds ACROSS ids THROUGH Tdrain TO Tsource;
  QUANTITY vdb ACROSS ibulk THROUGH Tdrain TO Tbulk;
  — .. other declarations
  QUANTITY VT : REAL :=VT0;
  — equations
  — ...
  — degradation equations (based on Hu model)
  — the time scale factor is inserted here
  IF Ids>0 USE
    VT'dot ==
      ageing_time_scale_factor *
      B*Ids*(Ibulk/Ids)**M;
  ELSE
    VT'dot == 0;
  END USE;
END ARCHITECTURE mos1

```

*Figure 14.3.* Transformation of a (uncompleted) MOS behavioural electrical VHDL-AMS model into a model for ageing simulation . The modified lines are written in **bold characters**. The degradation law ( $VT'_{dot} == \dots$ ) is given here in a naive form corresponding to the equation (14.2): in the real code, the  $I_{bulk}/Ids$  ratio is computed separately.

than  $10^{10}$  time steps! The simulation of the ageing would last as long as the real ageing using terabytes of memory.

To reduce the simulation size and duration, we use in simulation a dou-

ble time scale. The "electrical time" is the simulated time as given by the simulator and the "ageing time" is obtained by multiplying the simulated time by an ageing time scale factor. For instance, an ageing time scale factor equal to 1000 means that one second of simulated time represents one second of electrical time or 1000 seconds of ageing time. To introduce this time factor in the simulation, we use it as an acceleration factor: in each differential equation representing a degradation mechanism, the time first derivative of the evolving parameter is multiplied by the ageing time scale factor (see figure 14.3). The ageing time scale factor is given as a parameter of the model and must be the same for every degradation mechanism of all components. Finally, the value of the factor must be large enough to accelerate the simulation, but not too large because the parameters must evolve slowly during one period of the signals.

Using the previous degradable models of MOS transistors, we can simulate a circuit by "interconnecting" the degradable transistors. This method is more practical than the SPICE simulator with second software managing the degradation, but it is also limited to a few tenths of transistors.

## 5. Construction of the behavioural ageing model of a circuit

The third advantage of behavioural modelling is the possibility to build or to reuse a behavioural electrical model of a circuit or system to speed up the simulation. However, a major difficulty is to include the degradation effects in the behavioural model. In order to build such a behavioural ageing model, we have developed an original methodology based on a bottom-up analysis of the circuit: transistor level simulations are used to build the behavioural model of the circuit. To illustrate this method, we will use the example of the Operational Transconductance Amplifier (OTA) whose schematic is given in figure 14.2. The main parameter of the behavioural model is the transconductance  $G_M$ . For the demonstration, we will only consider the evolution of this parameter. In a real application, every significant behavioural parameter, such as the offset, the saturation levels of the output current, the input and output impedance, has to be taken into account.

### 5.1 Organisation of the behavioural ageing model

The behavioural degradable model of the OTA is built on the same principle as the degradable model of the transistor. The constant parameters of the models are transformed into variable quantities. To describe

the degradation mechanism, a degradation model is added that computes the evolution of the circuit parameters. This degradation model uses the inputs of the circuit to compute with a simplified model the drift of the parameters of the circuit (for instance  $G_M$ ). The critical step of the method is the construction of this circuit degradation model.

## 5.2 Principle of the construction of the degradation model of circuit

The method aims to build the ageing part of the behavioural model. The ageing of OTA circuit is due to the ageing of transistors. The MOSFET ageing is correlated to its bias conditions (its stresses). Therefore, the method is divided into four steps:

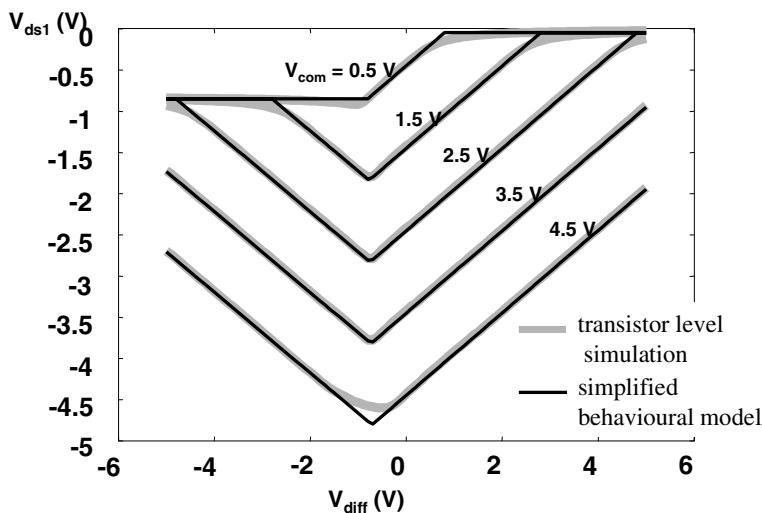
- the analysis of bias conditions of each transistor as a function of OTA bias conditions and construction of a simplified model,
- the assessment of each transistor ageing model that describes the parameter shift under the stress of bias conditions,
- the analysis of sensitivity of each electrical characteristics of OTA to each transistor parameter shift,
- the association and the simplification of the three relations to build the ageing part of the OTA behavioural model.

## 5.3 Bias conditions analysis

By electrical simulations, we can extract the whole electrical stresses of MOSFET transistors corresponding to each OTA electrical stresses conditions. The figure 14.4 shows the electrical voltage stresses  $V_{ds}$  of a transistor of the differential pair in case of several common mode voltages  $V_{com}$  as a function of the differential input voltage  $V_{diff}$  and fixed supply voltage conditions ( $V_{SS} = 0V, V_{DD} = 5V$ ). From this data, we extract a piecewise linear simplified model of MOSFET bias conditions as a function of bias conditions of OTA. The piecewise linear simplified model extraction is automated using an hybrid method based on evolutionary algorithms.

## 5.4 The transistor ageing model

All these electrical voltage stresses induce a physical degradation on the MOSFET electrical parameter and lead in long time use to MOSFET electrical characteristics shift. In our case study, all MOSFET devices of the OTA operate in saturation regime. This operating regime leads



*Figure 14.4.* Comparison between a transistor level simulation and the simplified behavioural model for the drain-source voltage of an input transistor as a function of the input differential voltage ( $V_{diff}$ ) and of the common mode voltage ( $V_{com}$ ).

to MOSFET degradation along time. To evaluate by simulation the effects of degradation, we use a transistor ageing model including the differential equation (14.2). The transistor ageing model is a critical step of this method (and of every other methods) because it highly depends on transistor technology and its parameters has to be extracted from experimental data. The association of the stress model and the transistor degradation laws used in ageing simulation generates cumulative effects to MOSFET threshold voltage parameter along the full ageing time scale.

## 5.5 Sensitivity analysis

The sensitivity analysis consists in computing the dependency of the electrical characteristics of the OTA to the threshold voltage shift due to the MOSFET ageing. This relation is obtained by electrical simulations. For each transistor, a simulation is performed forcing the threshold voltage shifting of transistor devices to a 20 mV step. The figure 14.5 shows the transconductance  $G_M$  shift of the OTA due to threshold voltage shifts of each PMOS devices. From these electrical simulations, we derive a linear relation between the threshold voltage shift and the behavioural model parameter of the OTA:

$$G_M = G_{M0} + \sum_n \frac{\partial G_M}{\partial V_{T,n}} \Delta V_{T,n} \quad (14.3)$$

where  $\Delta V_{T,n}$  is the  $n^{th}$  transistor threshold voltage shift due to the ageing.

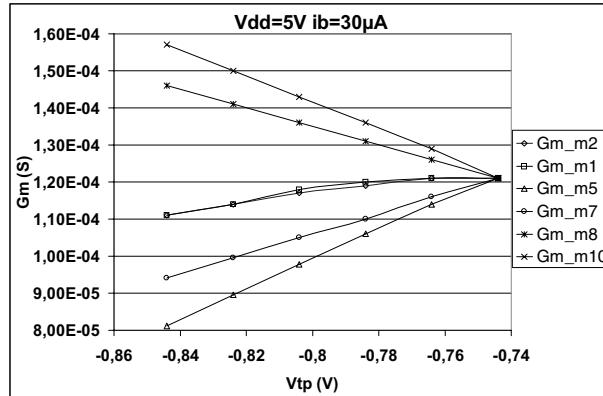


Figure 14.5. OTA transconductance shift due to  $V_T$  shift (transistor level simulations)

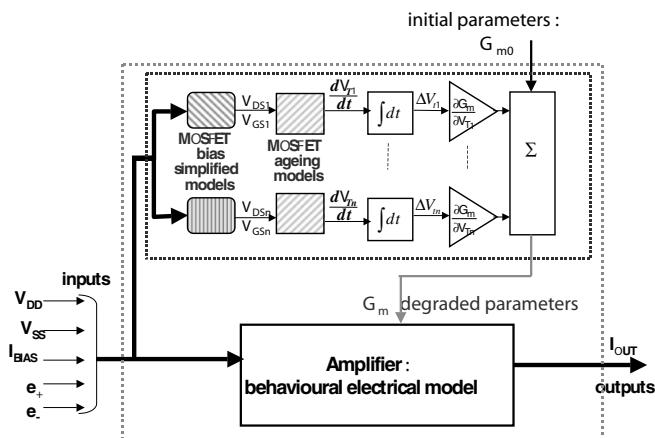


Figure 14.6. A functional schema of OTA ageing model.

## 5.6 An OTA ageing behavioural model

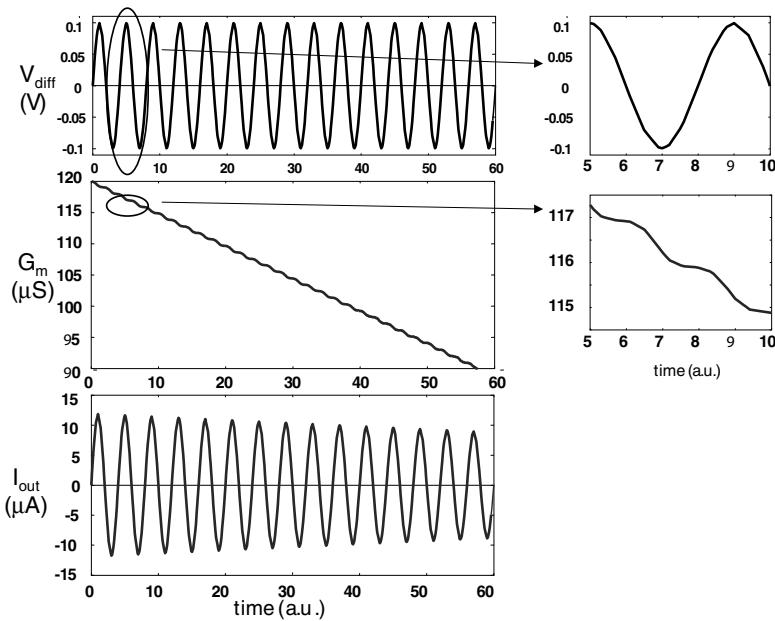
Based on previous results, the last step is to modify behavioural electrical model to include the ageing effects. This modification consists in replacing each behavioural parameter (for example  $G_M$ ) by a quantity defined by an equation like (14.3). Some internal quantities (here  $\Delta V_{T,n}$ ) are added those represent the ageing. They are modelled using the equation (14.2) and the simplified model linking the MOSFET bias conditions to the bias conditions of OTA. The figure 14.6 shows the functional blocks of the OTA ageing behavioural model that synthesises the method to build it. In this simplified diagram, only one behavioural parameter of the OTA (the transconductance  $G_M$ ) is degraded: a real model must degrade all the behavioural parameters.

## 5.7 Using the model for simulation

Once the behavioural degradable model of the circuit is built, it can be used to get the impact of the OTA ageing on a system performance. This point of view is illustrated by results obtained by electrical simulation of the OTA ageing model. The figure 14.7 shows results from an electrical simulation where the ageing of transistors due to hot carrier effects is taken into account over an arbitrary time scale. The OTA is biased to operate in the linear regime. This simulation show the decay of the transconductance of the OTA and its sensitivity to input voltages. The same simulation shows the effect of the degradation on the output current: its amplitude decrease.

## 6. Conclusion

We have presented an original method to build a behavioural ageing model of a circuit usable for fast reliability simulations of systems. This method uses transistor level electrical simulations of the circuit and a transistor ageing model. The resulting ageing model of the circuit makes possible reliability simulation taking into account the operating conditions dependent stresses of each transistor: by transient electrical simulations of the circuit in its system, the ageing of the electrical characteristic of the circuit can be analysed and its effect on the function of the system can be assessed. As the model is written in the standard behavioural modelling language VHDL-AMS, there is no need of a reliability specific simulator. Finally, the method for building an degradable behavioural model of a circuit can be used at higher level of description to build a degradable model of a system.



*Figure 14.7.* Simulation using the behavioural degradable model of the OTA. With a sinusoidal differential input voltage  $V_{\text{diff}}$ , we observe a decay of the transconductance  $G_M$  either directly or through the output current amplitude decrease. A detail shows the effect of the instantaneous value of the input signal on the decrease speed of the transconductance. The transistors degradation mechanism parameters used for computation of the behavioural model have been arbitrarily chosen.

## References

- [1] F. Jensen, "Electronic component reliability, fundamentals, modelling, evaluation and assurance", Willey & Sons, 1995
- [2] C. Hu, IC Reliability Simulation, IEEE J. Solid-State Circuits, vol. 27, No. 3, March 1992
- [3] <http://www.celestry.com>
- [4] M.M. Lunenborg, P.B.M. Wolbert, P.B.L. Meijer, T. Phat-Nguyen, J.F. VerWeij, "Press-A Circuit Simulator With Built-In Reliability Model For Hot-Carrier Degradation", ESREF 1993, pp157-161

## Chapter 15

# EXTENDING SYSTEMC TO ANALOG MODELLING AND SIMULATION

Giorgio Biagetti, Marco Caldari, Massimo Conti, and Simone Orcioni

*Dipartimento di Elettronica, Intelligenza Artificiale e Telecomunicazioni, DEIT*

*Università Politecnica delle Marche*

*via Brecce Bianche, 12*

*60131 Ancona, Italy*

*SystemC@deit.univpm.it*

**Abstract** This paper proposes a methodology for the extension of SystemC to mixed signal systems. An RF transceiver has been used to test the accuracy and stability of the algorithm proposed. Finally a simulation of a complex mixed-signal fuzzy controller is used to show the speed up achievable with a high level description of the analog block with respect to SPICE simulation.

**Keywords:** SystemC, mixed signal simulations, system level simulations.

### 1. Introduction

The rapid progress of VLSI process technology enables the implementation of complex systems in a single chip with several millions of gates. To deal with such complexity, parametric and reusable Intellectual Property (IP) cores, system level verification/validation techniques and a seamless design flow that integrates existing and emerging tools will be the key factors for a successful System on Chip (SoC) design.

In order to bridge the gap between technology capabilities for the implementation of SoCs and Electronic Design Automation (EDA) limitations, an environment integrating new concepts within new or existing languages and methodologies is necessary. SystemC [OSCI, online, Grotker et al., 2002], based on C++ language, is an emerging library and simulation environment used to provide a virtual model required to verify in short time the design specifications.

CAD vendors, designers and researchers are making a great effort in the development and standardization of SystemC, that is becoming a de facto standard. The implementation of digital and analog parts, such as DAC and ADC or RF frontends, in the same chip, requires a mixed-signal simulation. Research and standardization activities for the extension of SystemC to mixed signal systems are therefore necessary [Vachoux et al., 1997, Einwich et al., 2001, SC-AMS, online], as it has been done for VHDL-AMS and Verilog-A.

This work presents a possible solution for the extension to mixed-signal modelling of SystemC. The objective of this work is the definition of a methodology for the description of analog blocks using instruments and libraries provided by SystemC. In this way the analog and digital modules can be easily simulated together. The methodology allows the designer to describe the analog system either at a low or a high level using analog macromodels. Different Ordinary Differential Equations solvers (ODE) have been implemented in the SystemC code, including Euler and Adams-Bashforth multistep ODE solvers. An RF transceiver has been used to show in detail the methodology. Finally, the methodology has been applied to the modelling and simulation of a complex mixed-signal fuzzy controller.

## 2. Description of Analog Modules in SystemC

SystemC is a library of C++ classes developed to build, simulate and debug a System on Chip described at system level. SystemC provides an event driven simulation kernel and it describes the functionality of a system using processes. Three kinds of processes are available: methods, threads and clocked threads. A variation of a signal contained in a list, called sensitivity list and associated to each process, activates the process itself. The execution of threads may be suspended by calling the library function `wait(event)` and at the occurrence of `event` the thread will be resumed.

The proposed methodology uses methods and threads processes to model analog blocks in SystemC, in order to simulate analog blocks using the standard libraries and the simulation kernel of SystemC. This SystemC extension to mixed-signal systems enables the creation of a high level executable model of both digital and analog parts of a system in the same simulation environment. This allows a fast evaluation of the performances of the complete system.

In the following we will face the problem of implementing in SystemC continuous time lumped systems described by a system of non-linear

ordinary differential equations of the type

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ \mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{u}) \end{cases} \quad (15.1)$$

where  $\mathbf{f}$  and  $\mathbf{g}$  are vectors of expressions,  $\mathbf{x}$ ,  $\mathbf{y}$  and  $\mathbf{u}$  are state, output and input vectors, respectively.

Equation (15.1) can describe in detail the analog circuit under consideration or it may represent a high level macromodel describing its functionality. It does not include systems describable with a Differential Algebraic Equations system (DAE), however it is quite general. The SystemC libraries we introduce do not derive nor solve the Kirchhoff's equations from a SystemC description of the circuit: the user must describe the circuit in a model of the type reported in equation (15.1). This methodology allows the description of the analog block to be directly tractable by the SystemC simulation kernel, allowing a very efficient implementation. We think that a high level macromodel should be used in a SystemC system level simulation, using a SPICE-like simulator only when a detailed analysis is required.

One solution may be obtained using the explicit Euler formula

$$\begin{cases} \mathbf{x}(t + dt) = \mathbf{x}(t) + dt\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \\ \mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)). \end{cases} \quad (15.2)$$

The explicit Euler formula is not the most efficient solution, but it is the simplest, for this reason it has been chosen to explain the implementation in SystemC. The Adams-Bashforth formula is slightly more complex but shares a similar structure, differing only in the state update portion

$$\mathbf{x}(t + dt) = \mathbf{x}(t) + dt \sum_{k=0}^3 b_k \mathbf{f}(\mathbf{x}(t - k dt), \mathbf{u}(t - k dt)) \quad (15.3)$$

where

$$b_0 = +\frac{55}{24}, \quad b_1 = -\frac{59}{24}, \quad b_2 = +\frac{37}{24}, \quad b_3 = -\frac{9}{24}. \quad (15.4)$$

Both algorithms have been implemented, and the results will be shown in the following.

Different solutions are possible in the SystemC modelling of processes implementing (15.2) and (15.3) and in the choice of the time step  $dt$ :

1 A global constant time step for all the analog processes. This “analog simulation clock” activates all the analog blocks. This

solution is the simplest, but very inefficient since it requires a very small time step to reach an accurate solution.

- 2 A global adaptive time step for all the analog processes. All the analog blocks are activated by this adaptive “analog simulation clock”. A new global thread is necessary to calculate at each time instant the smallest time step required to guarantee the required accuracy. This thread acts as an analog simulation kernel. This solution is inefficient when analog blocks with different time constants are present in the same system.
- 3 The solution proposed in this paper is that each analog process calculates its adaptive time step, and it passes this time step to the blocks to which its outputs are connected. The analog inputs activate the process, that calculates the new state and output values. Then it sends them to the connected analog blocks after the minimum between its time step and an interval derived from the time steps received from the input blocks.

The SystemC implementation of an analog block representing a system of ordinary differential equations is reported in the following. The module is composed of two processes: the calculus thread and the activation method, one for each input module. The activation method (**sense**) starts when a change occurs in the signals coming from the corresponding input module. The **calculus** thread, that updates the state and output vectors, is activated only by signals coming from the activation method according to the algorithm reported in following pseudo SystemC code, simplified for a case in which the module has only one input module:

```
struct example : sc_module, analog_module
{
    sc_in <double> input, deltat_in;
    sc_out <double> output, deltat_out;

    virtual void field (double *) const;
    void calculus ();
    void sense ();

    SC_CTOR(example) : analog_module(...)
    {
        SC_THREAD(calculus);
        SC_METHOD(sense); sensitive << input;
    }
};
```

```

void example::sense ()
{
    double dt_ext = deltat_in;
    double in_new = input;
    if (abs(in_new - in) > THRESHOLD || dt_ext < dt / K)
        activation.notify();
    in = in_new;
}

void example::calculus ()
{
    x = 0;           // state initialization here.
    while (true) {
        step();      // inherited from analog_module.
                      // evaluate f in field(...) callback.
        output = x;  // evaluate g here.
        deltat_out = dt;
    }
}

void analog_module::step ()
{
    // evaluate f:
    field(direction);
    // adjust time step:
    double upper_limit = abs(x) * reltol + abstol;
    double lower_limit = upper_limit * mintol;
    if (abs(direction) * dt < lower_limit) {
        dt *= dt_factor;
        if (dt > dt_max) dt = dt_max;
    }
    while (abs(direction) * dt > upper_limit && dt > dt_min) {
        dt /= dt_factor;
        if (dt < dt_min) dt = dt_min;
    }
    // sleep (wait on activation event, timeout after dt):
    double t1 = sc_time_stamp().to_seconds();
    wait(dt, SC_SEC, activation);
    double t2 = sc_time_stamp().to_seconds();
    dt = t2 - t1; // (needed if wakened up by activation event)
    // update state:
    x += direction * dt;
}

```

The activation method calculates the difference between the current value of the analog input signal and the value at the previous step. The **calculus** thread is activated if this variation is bigger than a predefined reference threshold (**THRESHOLD**): this control can activate promptly the

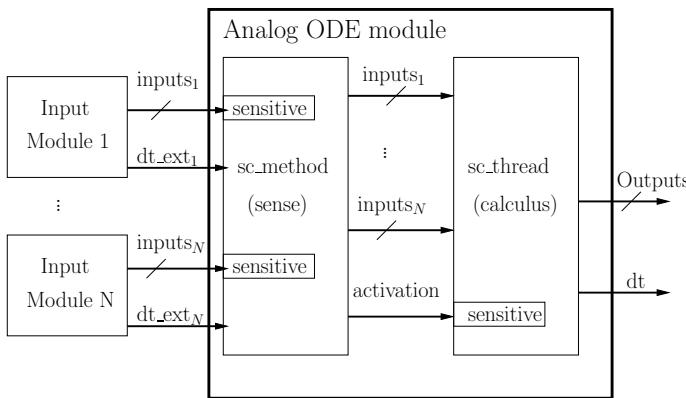


Figure 15.1. SystemC implementation of an analog block.

**calculus** thread when a sudden input variation occurs, avoiding delays or losses of impulse signals when the simulation step is too big.

The **calculus** thread is also activated if the minimum value between the time steps of the input modules is lower than the internal time step divided by a factor K. If K=1 the analog modules work with the same time step. If K>1 each analog module is allowed to work with a time step bigger than its input modules. When activated, the **calculus** thread calculates the state derivative (function **f** in equation (15.1) or function **field** in the SystemC code). Then the internal time step is adapted on the basis of the variation of the state value: the time step is decreased by a factor **dt\_factor**, if the variation is bigger than a threshold value (**upper\_limit**) and the time step is bigger than the parameter **dt\_min**. Conversely, the time step is increased by the same factor, if the variation of the state vector is lower than a threshold value (**lower\_limit**) and the time step is lower than the parameter **dt\_max**. In this way the internal time step increases when the state vector is reaching a steady state, and decreases when the state vector is rapidly changing. **dt\_max** and **dt\_min** fix the maximum and minimum values of the time step, respectively. Then the algorithm waits for an activation event coming from the activation method by using the SystemC `wait(timeout, e)` function. The thread execution is suspended waiting for the activation event **e** for a maximum time equal to the internal time step **dt**. After this time the thread is activated in any case. The state, output and time step are then updated. The values of the parameters **THRESHOLD**, **K**, **reltol**, **abstol**, **mintol**, **dt\_max**, **dt\_min** and **dt\_factor** strongly influence the accuracy of the simulation and the CPU

time required. An appropriate tuning has been performed and the following examples will show some results. A class library has been defined in order to make easy the creation of analog modules in SystemC.

### 3. Application Examples

Two examples will show the methodology used to model and simulate mixed-signal blocks in SystemC. An RF transceiver has been used to test the accuracy and stability of the proposed algorithm. A complex mixed-signal fuzzy controller is used to show the speed up achievable with a high level description of the analog blocks with respect to SPICE simulations.

#### 3.1 RF transceiver

The first example is based on the RF transceiver whose schematic is reported in Fig. 15.2. It is a high level description of a FSK modulator and demodulator. The transceiver is composed of the digital part generating the digital data to be transmitted (Baseband) with a 1 Mb/s rate, the VCO generating a sinusoid with a frequency of 2.410 GHz or 2.409 GHz depending on the transmitted bit, the Power Amplifier, the Channel with attenuation and noise insertion, the Low Noise Amplifier, the Mixer which multiplies the input signal with a reference sinusoid at 2.399 GHz, the Low Pass Filter, two Band Pass Filters centred at 10 and 11 MHz respectively, two Envelope Detectors, a Comparator and the synchronous digital Receiver. Different types of modules are used to describe the blocks: synchronous digital (D), analog (A), mixed (A/D and D/A), and analog modules described by ordinary differential equations (ODE) of the type (15.1). The digital signals are reported in thick lines in Fig. 15.2. The VCO evaluates a new value of the output sinusoid at each rising edge of the VCO clock. The analog modules (Power Amplifier, Channel, LNA and Mixer) perform instantaneous linear or nonlinear analog computations, therefore they can be described using simple SystemC methods sensitive to their analog input signals. The signal flow of these modules is therefore driven by the VCO clock. The LPF module models a simple linear filter described by the following linear differential equation

$$\begin{cases} u = \text{mixer\_out} \\ \dot{x} = \frac{u-x}{RC} \\ \text{LPF\_out} = x \end{cases} . \quad (15.5)$$

A SystemC module of the type reported in previous section is used to implement this block. Similar modules describe the behaviour of the Bandpass Filters and the Envelope Detectors used to discriminate the

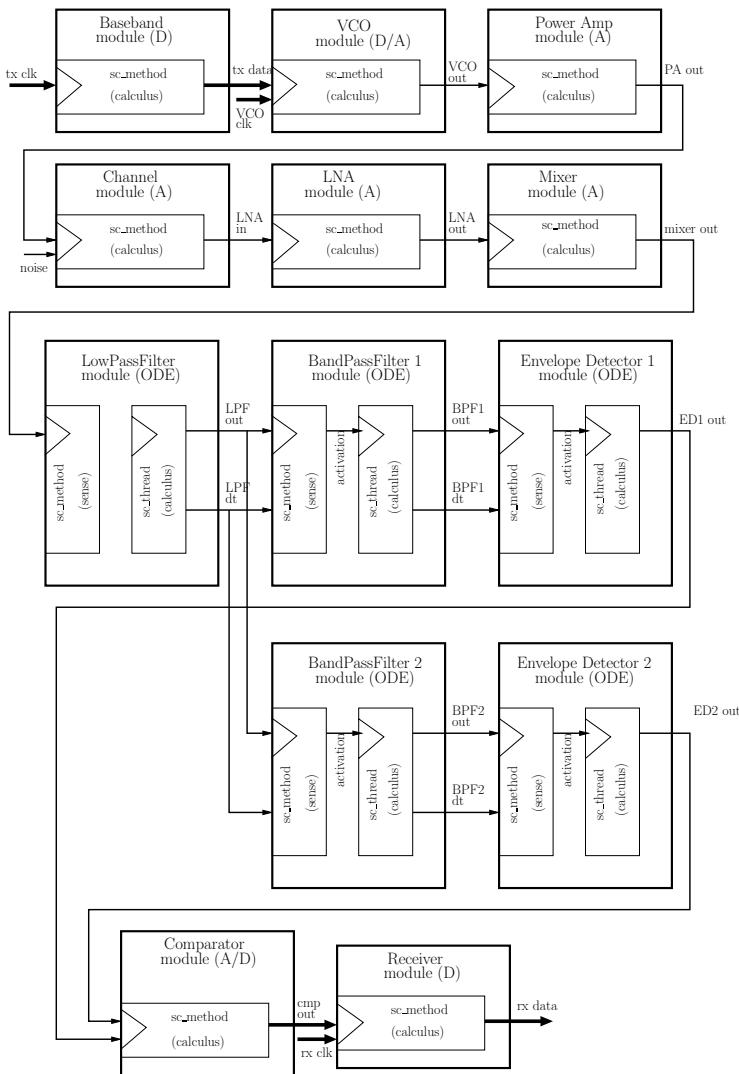


Figure 15.2. Mixed-signal SystemC architecture of the RF transceiver.

two frequencies. Equations (15.6–15.7) model the behaviour of the Band-pass Filters and the Envelope Detectors, respectively.

$$\begin{cases} u = LPF\_out \\ \dot{x}_1 = \frac{u-x_1}{RC} - \frac{x_2}{C} \\ \dot{x}_2 = \frac{x_1}{L} \\ BPF\_out = x_1 \end{cases} \quad (15.6)$$

$$\begin{cases} u = BPF\_out \\ \dot{x} = \frac{I_s}{C} \left[ \exp\left(\frac{u-x}{v_T}\right) - 1 \right] - \frac{x}{RC} \\ ED\_out = x \end{cases} \quad (15.7)$$

A 4 bit transmission ( $4 \mu s$ ) has been simulated in SystemC with different algorithms: global constant time step (“dt const in Table 15.1”), local adaptive time step with Euler formula (“euler”) and local adaptive time step with Adams-Bashforth formula (“adams”). Table 15.1 reports the value of the parameters of the algorithms used in the simulations, the CPU time required for the simulations and the accuracy of the simulation results. The accuracy is defined as the error of the waveform with respect to the value obtained with a simulation performed with a very small constant time step (0.05 ps) normalized to the maximum value achieved by the waveform. Table 15.1 reports the mean (NME) and maximum (NXE) value of this error, expressed in dB, for the outputs of the Band Pass Filter (BPF) and Envelope Detector (ED) modules. The global constant time step algorithm requires a very small time step (0.5 ps) to give accurate results, about  $-80$  dB. An increment in accuracy cause an exponential increment in CPU time. The local adaptive time step algorithms allow the modules whose signals change more slowly (BPF and ED), to have a larger local time step, thus allowing a reduction in CPU time. The Adams-Bashforth algorithm is more efficient than the Euler algorithm, in fact it reaches higher accuracy with lower CPU time. The results in Table 15.1 show that an error of  $-80$  dB is obtained with a gain in CPU time of a factor 4 with respect to the global constant time step algorithm (14.5 s against 61.1 s). A very accurate simulation of a 256 bit transmission ( $256 \mu s$ ) is obtained in about 1 hour of CPU time, 2 minutes if low accuracy is required. Fig. 15.3 shows the waveforms of the outputs of the BPFs, EDs and Comparator, and the internal time step of one ED during the simulation. The waveform of the RF signal, about 9600 sinusoid cycles, is not reported in the figure. The proposed implementation of analog modules in SystemC allows the analysis of the analog part of the RF transceiver in the same environment in which the digital part is described at a high level. The authors, for example, are working on the SystemC design of a Bluetooth transceiver from application layer to RF layer. This will make possible to carry out a system level power consumption analysis and a study of the effect of RF noise on the Bluetooth performances.

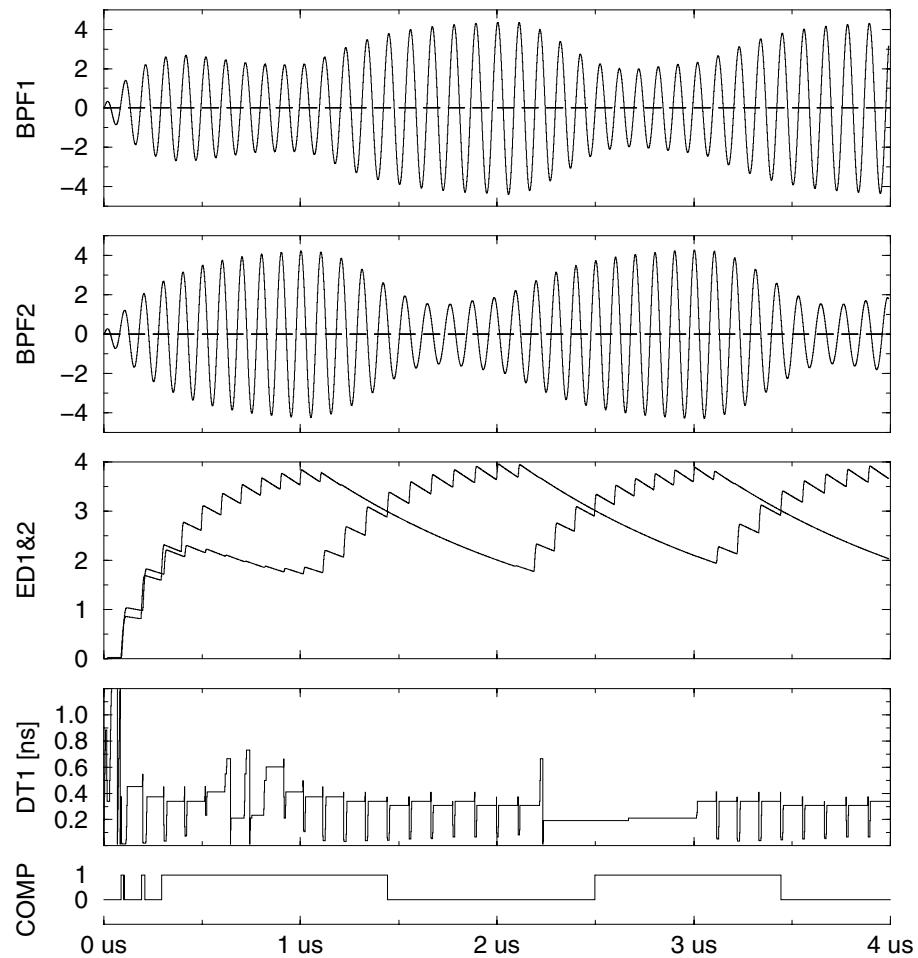


Figure 15.3. Waveforms of the outputs of Band Pass Filters, Envelope Detectors and Comparator, and the internal time step of one Envelope Detector during simulation.

### 3.2 Mixed-Signal Fuzzy Controller

The second example is a complex mixed signal fuzzy controller chip implemented in a 0.35- $\mu\text{m}$  CMOS technology [Orcioni et al., 2004, Conti et al., 2000]. The circuit implementing the 3-dimensional fuzzy partition membership function is the core of a mixed-signal Fuzzy Controller Integrated Circuit. This controller has been designed within a CNR project in collaboration with Merloni Elettrodomestici and Wr@p. The analogue implementation holds a key advantage over digital implemen-

method	VCO			CPU [s]	ED		BPF	
	clk [ps]	reltol	abstol		NXE <sup>1</sup> [dB]	NME <sup>2</sup> [dB]	NXE [dB]	NME [dB]
dt const	0.05			712.5				
dt const	0.1			336.5	-92	-98	-93	-102
dt const	0.5			61.1	-73	-79	-74	-83
dt const	1			34.0	-67	-72	-67	-76
dt const	10			3.0	-46	-52	-47	-56
dt const	100			0.5	-26	-31	-27	-36
euler	1	$10^{-4}$	$10^{-4}$	16.3	-57	-62	-59	-67
euler	1	$10^{-5}$	$10^{-5}$	19.3	-66	-72	-67	-76
adams	1	$10^{-5}$	$10^{-5}$	20.2	-94	-107	-80	-86
adams	5	$10^{-5}$	$10^{-5}$	14.5	-82	-99	-76	-83

<sup>1</sup> Normalized maXimum Error<sup>2</sup> Normalized Mean Error

*Table 15.1.* Performances of the different integration methods with varying accuracy parameters.

tation in applications where computational speed is needed. The chip is designed to be extremely flexible. In stand alone mode it is a fuzzy controller able to elaborate analog and digital signals and to communicate through an I2C bus with an EEPROM, necessary to store the parameters of the fuzzy system. In slave mode, the device acts as a peripheral of a microcontroller. The digital part, a microcontroller, an I2C driver and a register bank, is composed of 2864 standard cells, and the analog fuzzy engine is composed of 29673 MOSFETs. The analog part consists of 31 membership function blocks, one I/V converter, 3 V/I converters, 125 current multipliers, and 140 blocks of 5 bit D/A converters, used to store the parameters of the fuzzy engine. The fundamental block of the complete system, the membership function circuit, is reported in Fig. 15.4. It consists of current mirrors and 4 differential pairs. The complete system of 31 blocks, of the type reported in Fig. 15.4, implements 125 three-dimensional membership functions [Orcioni et al., 2004].

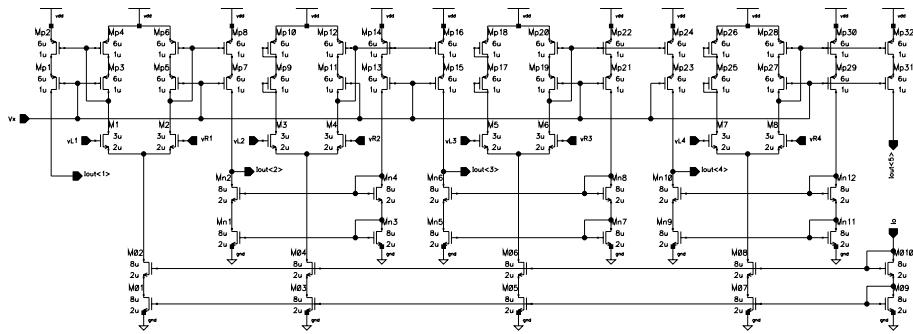
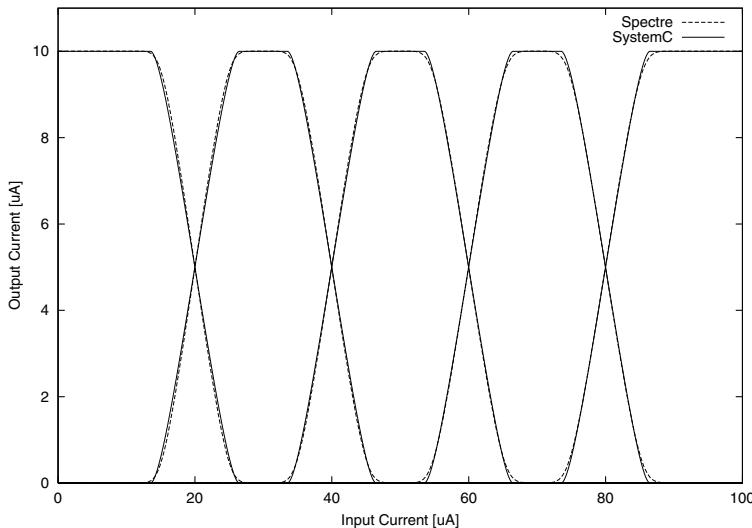


Figure 15.4. Membership Function Circuit.

The following formulae describe the dependence of the 5 output currents upon the input voltages

$$\begin{aligned}
 V_B &= \sqrt{\frac{2I_0}{\beta}} \\
 i_{OUTL} &= \begin{cases} 0 & v_D \leq -V_B \\ \frac{I_0}{2} + \frac{\sqrt{\beta}}{2} \frac{v_D}{2} \sqrt{4I_0 - \beta v_D^2} & -V_B < v_D < +V_B \\ \frac{I_0}{2} & +V_B \leq v_D \end{cases} \\
 i_{OUTR} &= I_0 - i_{OUTL} \\
 v_D &= v_L - v_R \\
 i_{OUT1} &= i_{OUTL1} \\
 i_{OUT2} &= i_{OUTR1} - i_{OUTR2} \\
 i_{OUT3} &= i_{OUTR2} - i_{OUTR3} \\
 i_{OUT4} &= i_{OUTR3} - i_{OUTR4} \\
 i_{OUT5} &= i_{OUTR4}
 \end{aligned} \tag{15.8}$$

These relationships have been implemented in the SystemC high level macromodel of the membership function block. In a similar way, all the analog and digital blocks have been described in SystemC at very high level, using nonlinear analytical models for each block and modelling the delay of analog blocks with an input and an output capacitance. Nevertheless, the functionality of the analog fuzzy engine is accurately described, as can be seen from Spectre simulations (dotted lines) and SystemC simulations (continuous lines) reported in Fig. 15.5. They are practically coincident. A transient simulation of the same circuit has been performed too, requiring 4h 38' in a spectre simulation and only



*Figure 15.5.* Five output currents of the first level membership function: DC spectre simulations (dotted lines) and SystemC simulations (continuous lines).

3.3 seconds in a SystemC simulation. This strong CPU reduction factor is due to the very high level description used in SystemC. In the first phase of the design, in which the chip architecture is not well defined and the correct functionality of the complete System on Chip must be verified, this speed up is very important. Furthermore, if necessary, a more detailed description of the analog circuit in SystemC is possible using the methodology presented in this work.

The implementation in the same simulation environment, SystemC, of both digital and analog parts enables a fast evaluation of the performances of the complete system. Furthermore, it is possible to model in SystemC the analog process to be controlled by the mixed-signal fuzzy chip. This allows the verification of the design in a real application.

## 4. Conclusion

The extension of SystemC to analog circuits is mandatory for the design of complex Systems on Chip including DAC or RF frontends. Many researches are starting in this field. This work proposes a simple and effective method to accomplish such an extension. The first results are encouraging in terms of accuracy of the simulation and CPU time required. The library developed, with an example, will be available at the authors web site ([www.deit.univpm.it](http://www.deit.univpm.it)).

**Acknowledgments.** This research has been sponsored in part by the European Medea+ project.

## References

- [Conti et al., 2000] Conti, M., Crippa, P., Orcioni, S., Turchetti, C., and Catani, V. (2000). Fuzzy controller architecture using fuzzy partition membership function. In *Proc. of 4th Int. Conf. on Knowledge-Based Intelligent Eng. Systems & Allied Technologies (KES2000)*, Brighton, UK.
- [Einwich et al., 2001] Einwich, K., Clauss, C., Noessing, G., Schwarz, P., and Zojer, H. (2001). SystemC extensions for mixed-signal system design. In *Proc. of Forum on Specifications & Design Languages (FDL'01)*, Lyon.
- [Grotker et al., 2002] Grotker, T., Liao, S., Martin, G., and Swan, S. (2002). *System design with SystemC*. Kluwer Academic Publisher.
- [Orcioni et al., 2004] Orcioni, S., Biagetti, G., and Conti, M. (2004). A mixed signal fuzzy controller using current mode circuits. *Analog Integrated Circuit and Signal Processing*, 38(2):215–231.
- [OSCI, online] OSCI, The Open SystemC Initiative (online). SystemC documentation. <http://www.systemc.org>.
- [SC-AMS, online] SC-AMS, SystemC-AMS Study Group (online). <http://mixsigc.eas.iis.fhg.de>.
- [Vachoux et al., 1997] Vachoux, A., Bergé, J. M., Levia, O., and Rouillard, J. (1997). *Analog and Mixed-Signal Hardware Description Languages*. Kluwer Academic Publisher.

IV

## LANGUAGES FOR FORMAL METHODS

*This page intentionally left blank*

As integrated circuits exceed a specific complexity, the analysis of their models becomes crucial so that additional care for complementary verification is required. In that context, formal specification and verification has become an attractive alternative. For example, equivalence checking and symbolic model checking are well investigated and widely accepted as complementary means to simulation during the last years. However, there are still several open issues, which have to be resolved for their wider applicability and acceptance. The LFM (Languages for Formal Methods) workshop of FDL'03 provided a forum for those issues mainly addressing formal languages and their application in formal verification. LFM covers the specification of systems and their functional properties, like Sugar/PSL, CTL, B, Kripke Structures, Process Algebras, Abstract State Machines in combination with their verification by simulation, model checking, and formal refinement, for instance.

The next 6 articles were selected from LFM workshop contributions and mainly cover the formal specification of state based systems and their refinement and verification.

In the first article, Börger gives a general introduction to the formal means of Abstract State Machines (ASMs) and outlines their principles by different applications, which were developed through the last years. The main focus of that article is on ASM specifications of state based systems, which can be easily transformed to verification of state based systems, e.g., for model checking.

Fischer et al. introduce a time extension for the  $\pi$ -calculus in the context of real-time system modelling. The extension basically corresponds to a time labelled transition system (TLTS) with real number labels at the transitions, which indicate progress in time. The industrial application of the extension mainly targets at the specification of Integrated Modular Avionics (IMA) systems and their design automation.

Boubekeur et al. present an approach, which starts from the formal specification by CHP (Communicating Hardware Processes). CHP are transformed to Petri-Nets and to the Verimag IF intermediate format to apply model checking and bi-simulation. The article gives performance studies of model generation and property checking times and closes with an application given by the example of a four-tap FIR filter.

Krupp et al. introduce the combined application of the RAVEN real-time model checker and B theorem prover for formal refinement. The focus is on the refinement of non-deterministic cycle accurate models to time-annotated model and on the definition of B patterns to support refinement automation. Their approach is outlined by the example of the echo cancellation unit of a mobile phone.

Romberg and Grimm consider the refinement of hybrid systems. They present a design flow with a formal refinement of hybrid systems. Their approach starts with a graphical HyCharts specification of the hybrid system. The discretized version of the HyCharts is a relaxed version with respect to time and taken as input to generate an equivalent SystemC model for simulation and synthesis.

We hope that the selected articles provide interesting and valuable insights to the current state of the art in the area of formal systems specification and verification.

Wolfgang Müller

*Universität Paderborn, Germany*  
*wolfgang@acm.org*

# Chapter 16

## LINKING ARCHITECTURAL AND COMPONENT LEVEL SYSTEM VIEWS BY ABSTRACT STATE MACHINES

Egon Börger

*Dipartimento di Informatica, Università di Pisa*

[boerger@di.unipi.it](mailto:boerger@di.unipi.it)

**Abstract** In hardware and software design model checkers are nowadays used with success to verify properties of system components [23]. The limits of the approach to cope with the size and the complexity of modern computer-based systems are felt when it comes to provide evidence of the trustworthiness of the entire system that has been built out of verified components. To achieve this task one has to experimentally validate or to mathematically verify the composition of the system. This reveals a gap between the finite state machine (FSM) view of model-checkable components and the architectural system view. In this paper we show how Abstract State Machines (ASM) can be used to fill this gap for both design and analysis, using a flexible concept of ASM component.

### 1. Introduction

Often model-checking and theorem proving are viewed as competing system verification methods, and as being in contrast to experimental validation methods which are based upon simulation and testing. However, all these methods are needed to cope with complex hw/sw systems, which need high-level models to exhibit the behavior that goes beyond what can be defined and analyzed by looking at the Finite State Machine (FSM) components. We explain in this paper how the framework of Abstract State Machines (ASMs) allows one to smoothly integrate current verification and validation methods into a design and analysis approach which permits to uniformly link abstract and detailed system views.

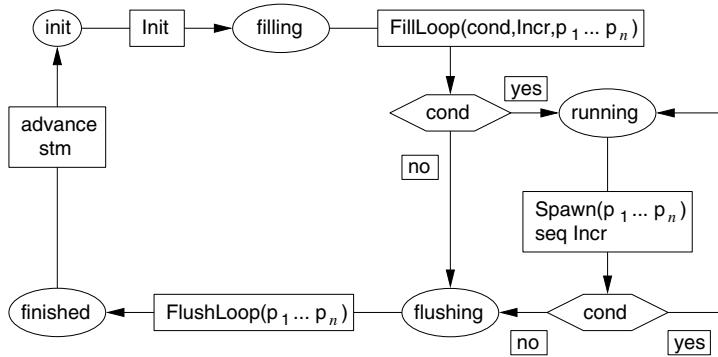


Figure 16.1. Control state ASM for SpecC pipe statements

ASMs can be introduced as a natural extension of FSMs, in the form of Mealy-ASMs or of control-state ASMs as defined in [9], namely by allowing a) *states* with arbitrarily complex or abstract data structures and b) *runs* with transitions where multiple components execute simultaneously (synchronous parallelism). Concerning the notion of run, *basic synchronous ASMs* come with the classical FSM notion of sequential run, characterized by sequences of successive computation steps of one machine. However, each single ASM step may involve multiple simultaneous actions. The example in Fig. 16.1, which is taken from [36], defines the top-level sequential structure of the execution semantics of so-called pipe statements in the language SpecC, an extension of C by system-level features which are used in industrial hardware design. These statements are parameterized by an *Initialization* statement, a *condition* which guards an iterative process, by an *Incrementing* statement used to advance the iteration, and by finitely many subprocesses which are spawned and deleted in a synchronized manner to fill, run and eventually flush the pipe. One finds numerous applications of such synchronous ASM models in hardware-software co-design, see [22, Ch.9] for a survey.

Asynchronously interacting FSMs, for example the globally asynchronous, locally synchronous Codesign-FSMs [35], are captured by asynchronous ASMs where runs are partial orders of moves of multiple agents. Multiple Mealy-ASM or control-state ASM components of an *asynchronous ASM* exhibit locally synchronous behavior, whereas the globally asynchronous system character is reflected by the generalization of the notion of sequential run to a partial order of execution steps of the components. For a simple example see the Production Cell ASM with six component machines in [17].

Concerning the extension of the notion of FSM-state, with ASMs the system designer has arbitrary structures available to reflect any given notion of state; we call this for short *freedom of abstraction*. This freedom of abstraction in combination with the powerful refinement notion offered by ASMs (see [11]) provide a uniform conceptual framework for effectively relating different system views and aspects in both design and analysis. In particular the framework supports a rigorous mediation between the architectural system view and the detailed view of the components out of which the system is built.

Based upon the extension of FSMs by ASMs built from FSM components, *model-checking* can be applied to the FSM components, as has been done using the connection of the SMV model checker to the ASM Workbench reported in [49],[25],[50]. To the global ASM, which describes the overall system built from the components, one can apply *theorem proving*, using as assumptions the model-checked local component properties. In addition *experimental validation* by simulation and testing can be applied to the high-level ASM as well as to its lower level components, due to the (mental as well as machine supported) executability of ASMs.

In Section 16.2 we explain how the abstraction and refinement mechanism coming with ASMs provides the flexibility that is needed for the navigation between different system levels and for a coherent combination of different system aspects. In Section 16.3 we show that a powerful component concept derives from a natural ASM submachine concept that unifies the architectural and the component level system view. It is based upon submachine replacement and refinement and provides various widely used composition schemes. For the benefit of the reader who does not know the notion of ASMs, in Section 16.2.1 we also provide a working definition which shows how ASMs capture in a natural and simple way the basic intuition of the concept of Virtual Machine. For further details, including numerous modeling and verification case studies and complete historical and bibliographical references, we refer to the AsmBook [22]<sup>1</sup>.

## 2. Relating high-level and component-level system views

To effectively and reliably link the architectural system view to its component-level view, and in general high-level models to more detailed ones, one needs two things:

- a uniform conceptual framework (read: a sufficiently general language) to express the notions involved at different levels of abstraction,

- a flexible method to navigate between the multiple levels of detailing.

Concerning the expressivity, from the short review we provide in Section 16.2.1 for the language of ASMs it should become clear that its few and simple basic constituents are general enough to integrate a great variety of useful system design and analysis patterns, leading from the requirements capture level down to the level of executable code. It is crucial that with the ASM method the same language can be used to define and analyse the various constituents of models, at all levels: the rules for the dynamics, whether for the global system or for its components, as well as the auxiliary procedures or functions on the variety of underlying data structures. In Section 16.2.2 we explain how the ASM refinement method allows one to cross those multiple abstraction levels in a way that coherently and organically links them and makes these connections documentable.

## 2.1 The language of ASMs

The language of ASMs is the language of mathematics, the language par excellence of any scientific discourse. It is the same language where ASMs and their constituents are defined and analysed, using appropriate notational conventions to achieve simplicity. As machines exhibiting dynamic behavior ASMs are as general as virtual machines and algorithms can be, due to their definition as transitions systems that transform abstract states. In fact the *states* of ASMs are arbitrary mathematical *structures*: data are abstract objects (read: elements of sets or instances of classes) equipped with basic operations and predicates (attributes or relations). A familiar view of such structures is to treat them as given by tables. The entries of such tables are called *locations* and come as pairs of a function or predicate name  $f$  and an argument  $(v_1, \dots, v_n)$ , which is formed by a list of parameter values  $v_i$  of whatever types. These locations represent the abstract ASM concept of basic object containers (memory units), which abstracts from any specific memory addressing and object referencing mechanism.

Location-value pairs ( $loc, v$ ) are called *updates* and represent the basic units of state change in ASM computations. In fact ASMs transform abstract states by multiple simultaneous conditional updates that represent control-structure-free “If-Then” directives, the most general form of virtual machine instructions. Technically speaking these instructions come as finite set of ordinary transition *rules* of the following general form

**if Condition then Updates**

where the guard *Condition* under which a rule is applied is an arbitrary expression evaluating to *true* or *false*. *Updates* is a finite set of assignments of the form

$$f(t_1, \dots, t_n) := t$$

whose simultaneous execution is to be understood as redefining the values of the indicated functions  $f$  at the indicated arguments to the indicated values. More precisely in the given state, for each rule with true guard, first all parameters  $t_i, t$  are evaluated to their values, say  $v_i, v$ , then the location  $(f, (v_1, \dots, v_n))$  is updated to  $v$ , which represents the value of  $f(v_1, \dots, v_n)$  in the next state.

Thus ASMs represent a form of “pseudo-code over abstract data” where the instructions are guarded function updates. The abstract understanding of memory and memory update allows the application domain modeler or the system designer to combine the operational nature of the concepts of location and update with the freedom of tailoring them to the level of detail which is appropriate for the given design or analysis task. The simultaneous execution of multiple updates provides a rather useful instrument for high-level design to *locally describe a global state change*, namely as obtained in one step through executing a set of updates of some locations. The local description of global state change implies that by definition the next state differs from the previous state only at locations which appear in the update set. This basic parallel ASM execution model easens the specification of macro steps (using refinement and modularization as explained below), it avoids unnecessary sequentialization of independent actions and it helps to develop parallel or distributed implementations. The synchronous parallelism is enhanced by a notation for the simultaneous execution of a rule  $R$  for each  $x$  satisfying a given condition  $\phi$ :

**forall**  $x$  **with**  $\phi$   
 $R$

We illustrate this here by the ASM rule defined in [14] for the Occam instruction that spans subprocesses: in one step the currently running process  $a$  creates  $k$  new subprocesses, activates them and puts itself to sleeping mode, where the process activation provides the current environment and positions each subagent upon the code it has to execute. In this example we use the object-oriented notation to denote the instantiation of some of the state functions.

```
OCCAMPARSPAWN =
if  $a.mode = running$  and  $instr(a.pos) = par(a, k)$  then
```

```

forall  $1 \leq i \leq k$  let  $b = new(Agent)$  in
  ACTIVATE( $b, a, i$ )
   $parent(b) := a$ 
   $a.mode := idle$ 
   $a.pos := next(a.pos)$ 
where ACTIVATE( $b, a, i$ ) =
   $\{b.env := a.env, b.pos := pos(a, i), b.mode := running\}$ 

```

Similarly, non-determinism as a convenient way to abstract from details of scheduling of rule executions can be expressed by rules of the form

**choose**  $x$  **with**  $\phi$   
 $R$

where  $\phi$  is a Boolean-valued expression and  $R$  is a rule. The meaning of such an ASM rule is to execute rule  $R$  with an arbitrary  $x$  chosen among those satisfying the selection property  $\phi$ .

In defining an ASM one is free to choose the abstraction level and the complexity and the means of definition of the auxiliary functions, which are used to compute locations and values in function updates. The following standard terminology, which is illustrated by Fig. 16.2 and is known as classification of ASM functions, names the different roles these functions can assume in a given machine  $M$  and provides a strong support to modular system development.

*Static* functions never change during any run of  $M$  so that their values for given arguments do not depend on the states of  $M$ , whereas

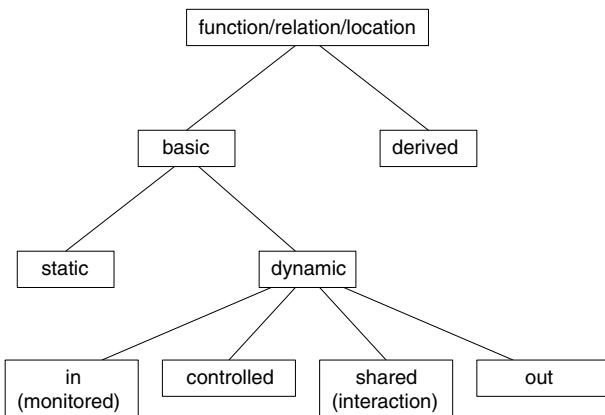


Figure 16.2. Classification of ASM functions, relations, locations

*dynamic* functions may change as a consequence of updates by  $M$  or by the environment, so that their values for given arguments may depend on the states of  $M$ . Defining the static functions is independent from the description of the system dynamics. This supports to separate the treatment of the statics of a system from that of its dynamic behavior to reduce the complexity of the overall development task. Furthermore, whether the meaning of these functions is determined by a mere signature (“interface”) description, or by axiomatic constraints, or by an abstract specification, or by an explicit or recursive definition, or by a program module, depends on the degree of information-hiding the specifier wants to realize.

Dynamic functions can be thought of as a generalization of array variables or hash tables. They are divided into four subclasses, depending on whether the machine or its environment (in general: other agents) update the function in question. *Controlled* functions (of  $M$ ) are dynamic functions which are directly updatable by and only by  $M$ , i.e. functions  $f$  which appear in at least one rule of  $M$  as the leftmost function (namely in an update  $f(s) := t$  for some  $s, t$ ) and are not updatable by the environment. These functions are the ones which constitute the internally controlled part of the dynamic state of  $M$ . *Monitored* functions are dynamic functions which are read but not updated by  $M$  and directly updatable only by the environment. They appear in updates of  $M$ , but not as the leftmost function of an update. These monitored functions constitute the externally controlled part of the dynamic state of  $M$ . To describe combinations of internal and external control of functions, *interaction* functions are used, also called *shared* functions and defined as dynamic functions which are directly updatable by the rules of  $M$  and by the environment and can be read by both. *Out* functions are dynamic functions which are updated but not read by  $M$  and are monitored by the environment.

The concepts of monitored, shared and out(put) functions allow one to separate in a specification the computation concerns from the communication concerns. The definition does not commit to any particular mechanism (e.g. message passing via channels) to describe the exchange of information between an agent and its environment. The only (but crucial) assumption made is that in a given state the values of all functions are determined.

Another pragmatically important distinction is that between *basic* and *derived* functions. Basic functions are functions which are taken for granted (declared as “given”); derived functions are functions which even if dynamic are not updatable either by  $M$  or by the environment, but may be read by both and yield values which are defined by a fixed scheme

in terms of other functions. Thus derived functions are a kind of global method with read-only variables; to orthogonalize system descriptions one can specify and analyze them separately from the main machine.

Many other system design patterns can be conveniently integrated into ASMs via the two above classifications of functions or analogously locations (see [13] for details). For example if an object-oriented notation is desired, one can naturally support it by parameterizing functions  $f(x)$  (similarly for rules representing methods) by a parameter  $a$  denoting an object (or an agent), thus creating ‘instances’  $\lambda x\ f_a(x)$  of  $f$ . As example see the relativization of the state functions *mode*, *pos*, *env* in the machine OCCAMPARSPAWN above. An illustration for introducing rule instances is given below (FIFO-buffer example in Section 16.3).

It is pragmatically important that the definitions are all phrased in terms of basic mathematical or algorithmic notions and thus can be grasped by every system engineer, relying only upon fundamental operational concepts of computing that are familiar from programming practice, independently from the knowledge of any specific computational platform or programming/design language or underlying logic. However, also a detailed formal definition of the semantics of ASMs, which is useful as basis for advanced analysis methods and for the development of well-documented tool support for ASMs, is available in textbook form in the AsmBook [22, Ch.2]. It is formulated there in terms of a first-order-logic-based derivation system.

## 2.2 Navigation between levels of detail

The method of ASMs exploits various means to provide smooth but rigorous links between different levels of detail in a system description. As explained in the previous section, already the language of ASMs via the function classification supports to separately describe the statics of a system and its dynamics<sup>2</sup>, similarly to separately specify the internal behavior of a component and its interface to the environment—and to link them together in the overall ASM model. It also supports to smoothly incorporate into a system model specific language constructs, e.g. procedural or purely functional or logico-axiomatic descriptions or even full-fledged programs of a given design or programming language, all of which can be given separately from the ASM rules where they are used.

The natural companion of abstraction is refinement. Thus it should not come as a surprise that in the same way in which ASMs provide maximal freedom of abstraction, as explained above, they also provide a *most general notion of refinement*, a kind of meta-framework which

can be tailored to given practical needs of componentwise system development. This framework integrates well-known more specific notions of refinement which have been defined in the literature (see [11, 39] for details). More importantly it supports to pass in a well-documented and checkable manner from an abstract model to a more detailed model, in a chain of such models, thus linking the outcome of the different phases of system design and making these links communicatable to the various experts involved. Here is a list of the main levels and experts involved.

- *Different levels of precision for accurate high-level human understanding* of a system. The language of ASMs allows one to fine-tune the models to the needed degree of rigour. Numerous experts are involved who speak different languages, think at different levels of abstraction and yet have to come to a common understanding of a system:
  - Application domain expert and requirements engineer. The natural interpretation of ASMs as control-structure-free system of “If-Then” directives supports their successful use for building *ASM ground models* which allow one to rigorously capture system requirements and help ensuring a correct understanding of the resulting model by all parties involved. Since some stakeholders, for example the application domain experts, are usually not familiar with the principles and concepts of software design, it is crucial that the language of ASMs permits to formulate ground models in terms of the application domain so that one can recognize in the model the structure of the real-world problem. The language of ASMs allows one to combine in a uniform framework data model features, which are typically of conceptual and application oriented nature, with functional features defining the system dynamics and with user-interface features defining the communication between the data and functional model and neighboring systems or applications. The systematic representation of the requirements in ASM ground models provides a basis for requirements inspection and thus makes the correctness and the completeness of the requirements checkable for all the stakeholders. Due to the rigorous character of ASM models this may include the verification of overall system properties. Due to the executable character of ASMs, via ASM ground models one also obtains a possibility for early system validation, typically by simulation of user scenarios or of components. In addition, the frugal character of the

ASM language helps to cope effectively with the continuous change of requirements by adaptations of the ground model abstractions, see [10].

- Requirements engineer and system designer. Here ground models play the role of an accurate system blueprint where all the technical essentials of the software system to be built are layed down and documented as basis for the high-level design decisions.
  - System designer and programmer. The component definition in Section 16.3 shows how ASMs mediate between the overall system view defined by the designer and the local view of model-checkable components developed by the programmer.
  - System designer or programmer and tester or maintenance expert. The hierarchy of ASM models leading from the ground model to code allows one to identify the place where something may or did go wrong or where a desired extension can be realized appropriately. From the sequence of models and from the descriptions of model-based runtime assertions appearing in the test reports one can read off the relevant design features to locate bugs or trace versioning effects.
  - System designer and system user. The abstractions built into ASM ground models help to provide an understanding of the overall functionality of the system, to avoid erroneous sytem use.
- *Human understanding and implementation.* These two levels can be linked by an organic, effectively maintainable refinement chain of rigorous coherent models. At each level the divide and conquer method can be applied to prove a system property for the detailed model, namely by proving it from appropriate assumptions in the abstract model and showing that the refinement is correct and satisfies those assumptions. For this the practicality of the ASM refinement method is crucial, namely for faithfully reflecting an intended design decision (or reengineering idea) and for justifying its correct implementation in the refined model. The practicality stems from the generality and flexibility of the ASM refinement notion, as explained in [11]. By providing this possibility the ASM method fills a gap in the UML framework.
  - *Design and analysis.* This well-known separation of different concerns (“You can analyze only what has been defined already”) helps not to restrict the design space or its structuring into components

by proof principles which are coupled to the design framework in a fixed a priori defined way. The ASM method permits to apply any appropriate proof principle (see the discussion below) once the model is defined. A typical class of examples where this distinction is crucial is the rigorous definition of language or platform standards. An outstanding ASM model for such a language standard is the one built for SDL-2000 in [31]. Another example is provided by the ASM model for C# defined in [15, 44] to reflect as much as possible the intuitions and design decisions underlying the language as described in the ECMA standard [28] and in [33]. An example for the combination of an ASM definition and of its detailed mathematical analysis has been developed in [46] for Java and its implementation on the Java Virtual Machine. The model is the basis for a detailed mathematical analysis, including proofs that Java is type-safe, that the compiler is correct and complete and that the bytecode verifier is complete and sound.

- *Different analysis types and levels.* Such a distinction is widely accepted for system design levels and methods, but only rarely is it realized that it also applies to system analysis. The language of ASMs allows one to calibrate the degree of precision with respect to the needs of the application, whether data oriented (e.g. using the entity relationship model) or function oriented (e.g. using flow diagrams) or control oriented (e.g. using automata of various kinds). Here are the major levels of analysis the ASM method allows one to treat separately and to combine their results where needed.
  - Experimental *validation* (system simulation and testing) and mathematical *verification*. For example the *ASM Workbench* [24] has been extensively used in an industrial reengineering project at Siemens [18] for testing and user-scenario simulation in an ASM ground model. The *AsmGofer* system [41, 42] has been used in connection with the ASM models for Java and the JVM in [46] for testing Java/JVM language constructs. In a similar way the .NET-executable *AsmL* engine [29] is used at Microsoft Research for ground model validation purposes. The possibility of combining where appropriate validation with verification in a uniform framework helps not to be at the mercy of only testing.
  - *Different verification levels* and the characteristic concerns each of it comes with. Each verification layer has an established degree of to-be-provided detail, formulated in an ap-

propriate language. E.g. reasoning for human inspection (design justification by mathematical proofs) requires different features than using rule-based reasoning systems (mechanical design justification). Mathematical proofs may come in the form of proof ideas or proof sketches (e.g. for ground model properties like in [7],[17],citeBoRiSc00) or as completely carried out mathematical proofs (see for example the stepwise verification of a mutual exclusion protocol [16] or of the compilation of Occam programs to Transputer code [14] that is split into 16 refinement proofs). Formalized proofs are based on inference calculi which may be operated by humans (see for example various logics for ASMs which have been developed in [32],[43],[38],[30],[45],[37]) or as computerized systems (see for example the KIV verification [40],[39] of the ASM-based WAM-correctness proof in [20] or the ASM-based PVS-verification of compiler back-ends in [26],[27],[30]). For mechanical verification one has to distinguish interactive systems (theorem proving systems like PVS, HOL, Isabelle, KIV) from automatic tools (model checkers and theorem provers of the Otter type). Each verification or validation technique comes with its characteristic implications for the degree of detail needed for the underlying specification, for the language to be used for its formulation and for the cost of the verification effort. Importantly, all these techniques are integratable into the ASM framework.

We resume the above explanations by Fig. 16.3 which is taken from [19] and illustrates the iterative process to link ASM specifications to code via model-supported stepwise design.

### 3. Submachine-based component concept

We present in this section a component concept for ASMs that goes beyond the concept of components of an asynchronous ASMs. In asynchronous ASMs multiple synchronous ASM components are put together to a globally asynchronous system with partial order runs or interleaving runs instead of sequential runs. A simple example is given in Section 16.3.3, more involved examples are provided by the class of globally asynchronous, locally synchronous Codesign-FSMs [35] mentioned in the introduction. In Section 16.3.1 we extend this notion of ‘ASM component’ by providing some widely used operators to compose ASMs out of submachines. These operators maintain the atomic-action-view which is characteristic for the synchronous parallelism of basic ASMs. Other

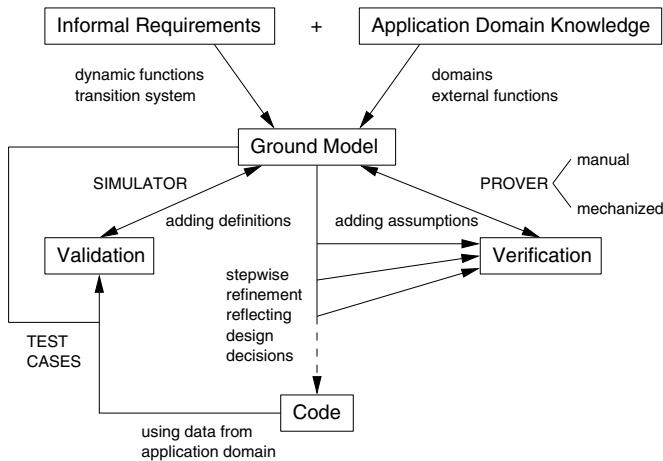


Figure 16.3. Models and methods in the development process

operators allow one to build ASMs out of components with structured or durative ‘actions’ and are surveyed in Section 16.3.2.

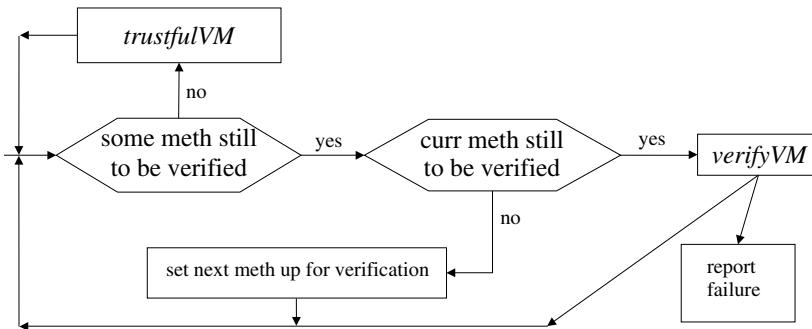
### 3.1 Operators for the Composition of Components

Since the dynamics of ASMs is defined by rules, the most general form of an ASM component is that of a rule<sup>3</sup>. Therefore the most general form of composition of a complex ASM out of a simpler one is by *rule replacement*, namely of a (simpler) *rule* occurring as subrule of say *M* by another (more complex) *rule'*. The result of such a substitution of *rule'* for *rule* in *M* is written *M(rule'/rule)*. The ASM method leaves the freedom to the designer to impose further restrictions on the form of the component rules or on their use, guided by the needs of the particular application. We survey some examples in Section 16.3.2.

We discuss now some particularly frequent special cases of rule replacement which are used to build complex machines out of components. **Macro refinement.** A typical use of rule replacement is made when in a sequence of stepwise model refinements it comes to *replace a macro definition* by a new one. For this case we use the following special notation where *Redef(macro)* stands for the underlying new definition of *macro*.

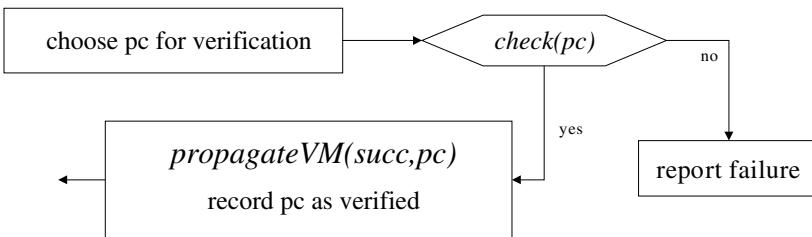
$$M(\mathbf{redef} \, macro) = M(Redef(macro)/macro)$$

**Rule refinement.** Another frequent use of rule replacement occurs upon the *refinement of an abstract rule by a concrete ASM*, a generalization of macro refinement in case the macro is a rule. For example



verifyVM built from submachines propagate, succ, check

Figure 16.4. Building diligent JVM from components trustfulVM and verifyVM



$succ_I \subset succ_C \subset succ_O \subset succ_E$  and  $propagate_I \subset propagate_E$

Figure 16.5. Building verifyVM from components propagateVM, check, succ

the abstract submachines which appear in Fig. 16.1 can be defined in more detail as independent modules, see [36]. Another example can be found in the sequence of stepwise refined models for the Java Virtual Machine in [46]. There, one ASM defines a diligent JVM by adding to a bytecode interpreter component *trustfulVM* a bytecode verifier component, which calls method per method an abstract rule *verifyVM*, as illustrated in Fig. 16.4.

The abstract rule *verifyVM* is then refined by a submachine *verifyVM* which successively checks every instruction to satisfy the necessary type conditions and in the positive case propagates them to all successor instructions, using three components *check*, *propagate*, *succ* as illustrated in Fig. 16.5. For this form of refinement one can reuse the notation introduced for macros, writing **redef**  $M = \text{Redef}(M)$  where  $\text{Redef}(M)$  stands for the refinement of  $M$ .

**Adding/deleting rules.** Another common case of submachine replacements is the *addition or deletion of rules*. Due to the synchronous parallelism of ASMs no special notation is needed for adding rules, whereas for the deletion of  $R$  from  $M$  we write  $M$  **minus**  $R$ . Adding new rules is characteristic of purely incremental (also called conservative) refinements. A recent example is provided in [15, 44] to structure C# into layered components of by and large orthogonal language features, similarly to the decomposition in [46] of Java and the JVM into a hierarchy of components, as illustrated for the components *propagateVM*, *check*, *succ* of the JVM verifier in Fig.16.5. The model for the entire language is a parallel composition of all submachines. This incremental system development allows one to successivly introduce the following language layers, a feature one can exploit not only for a piecemeal validation and verification of the entire language (as done for Java and its implementation on the JVM in [46]), but also for a systematic approach to teaching a complex programming language to beginners (see [12]):

- *imperative core*, related to sequential control by while programs, built from statements and expressions over the simple types of C#,
- *static class features* realizing procedural abstraction with class initialization and global (module) variables,
- *object-orientation* with class instances, instance methods, inheritance,
- *exception handling*,
- *concurrency*,
- *delegates* together with events (including for convenience here also properties, indexers, attributes),
- *unsafe code with pointer arithmetic*.

**Adding guards/updates.** The refinement of rules is often defined at the level of the two constituents of rules, namely guards and updates. For a machine which executes a given machine  $M$  only if a certain guard condition  $G$  is satisfied we write

**addGuard  $G$  to  $M$  = if  $G$  then  $M$**

For adding a set of updates to the updates of a rule in ‘normal form’ **if  $Cond$  then  $Updates$**  we write as follows:

**$M$  addUpd  $U$  =  
let  $M$  = if  $Cond$  then  $Updates$  in**

```

if Cond then
    Updates
    U

```

In Section 16.3.3 we illustrate the use of these two operators by a small example for a componentwise system definition.

### 3.2 Specific ASM component concepts

In this section we briefly review some more specific component concepts which have been defined in terms of ASMs and have been implemented using some of the major systems for the execution of large classes of ASMs.

All the operators defined in Section 16.3.1 maintain the atomic-action-view which is characteristic for the synchronous parallelism of basic ASMs. In [21] three fundamental standard operators are defined for ASMs with non-atomic behavior, namely sequential composition, iteration and recursive submachines. These three concepts have been implemented in the publicly available system AsmGofer [41] for executing ASMs. The definitions guarantee that one can build global ASMs, out of component ASMs, whose synchronous parallel computation may be viewed both in a global atomic way and in a local non-atomic way. For each of these three operators, the global view of the complex machine treats the component as a black-box that performs an atomic action. The global properties can be formulated on the basis of the analysis the local view allows one to perform, namely by looking in a white-box manner at the internal details of the non-atomic (iterative or otherwise durative) behavior of the component. This double view allows one to perform a hierarchical system development, where the model checking of the components is the basis for a global system analysis by mathematical proof and experimental validation (simulation) techniques.

In [47] an ASM component concept is defined which treats components in terms of the services they export or import.

The open source XASM tool [1] for the execution of ASMs is built around a component-based C-like language. It has been used in [4] for the implementation of component interaction. It also implements the above mentioned notion of sequential submachines. However, it provides no support for hierarchical modeling.

An specific ASM component concept that follows the pattern of Mealy the high-level verification of VHDL-based hardware design. It comes with a component-based verification technique that has been used in two industrial projects at Siemens reported in [42]. Roughly speaking, a component is defined there as a basic ASM whose rule may contain

submachine calls and whose interface consists of VHDL-like in/out functions; they are used for providing input respectively output at successive computation steps to compute input/output behaviour in the way known from Mealy FSMs. Composition of such components is defined not by name sharing of input and output functions, but by connecting inputs and outputs: several inputs may get the same input value; a connected input is connected either with an ouput or with an input which in the transitive closure is not connected to the original input (so that one can determine for connected inputs the input value determining source).

The tool AsmL [29] developed at Microsoft is used in [6, 5] to implement on the .NET platform behavioral interface specifications by ASMs, including component interfaces, and to test COM components against these specifications.

### 3.3 Componentwise system development: an example

We illustrate the above two operators for adding guards and updates by a small example taken from [34, Ch.3]. We use the occasion to also show the above mentioned painless introduction of object-oriented notation by parameterization of rules. The example deals with a stepwise definition of a bounded FIFO buffer from a basic component for a 2-phase handshake protocol.

The first step consists in defining this protocol to transmit a data *value* from a sender to a receiver who has to acknowledge the receipt before the next data value can be sent. Sending is possible only if *ready* = *ack* (first handshaking), the action includes flipping the boolean variable *ready*. An acknowledgement can be made only if *ready* = *ack* is false (second handshaking), the action includes flipping the boolean variable *ack*. This is expressed by the following ASM which defines a channel with rules for sending and receiving data, using a submachine FLIP to flip the value of a boolean variable. The sending rule is parameterized by the data to be sent.

```

CHANNEL = {SEND, RECEIVE} where
  SEND(d) = if ready = ack then
    FLIP(ready)
    val := d
  RECEIVE = if ready ≠ ack then FLIP(ack)

```

The second step consists in parameterizing this handshake protocol machine by an agent **self**, which can be instantiated to produce independent copies of the machine. Since the state of CHANNEL is made out of the three 0-ary functions *ready*, *ack*, *val*, also these functions have to be relativized to **self**. This comes up to replace CHANNEL

by **self** .CHANNEL, which in ordinary mathematical notation is written CHANNEL(**self**); similarly for SEND, RECEIVE and the functions *ready*, *ack*, *val*.

The third step consists in defining an unbounded FIFO-buffer from two instances of CHANNEL as its components, one for placing an input into the queue and one for emitting an output from the queue. Since appending an element to the queue and deleting from it an element are largely independent from each other, we want to avoid a possibly premature design decision about the particular scheduling for these two operations. As a consequence the FIFO-buffer is formulated by an asynchronous ASM, which consists of two synchronous ASM components built from instances of CHANNEL: the instantiation FIFOBUFFERIN by an agent *in* appends elements to the queue, the instantiation FIFOBUFFER OUT by an agent *out* deletes elements from the queue.

When an element is sent that should be put into the buffer, upon acknowledging the receipt channel agent *in* has to append the received value to the (tail of the) queue. This is achieved by extending the set of updates performed by *in*.RECEIVE.

$$\begin{aligned} \text{FIFOBUFFERIN} &= \{\textit{in}.SEND, \text{BUFRCV}\} \text{ where} \\ \text{BUFRCV} &= \textit{in}.RECEIVE \textbf{addUpd APPEND}(\textit{in}.val, \textit{queue}) \end{aligned}$$

When an element is sent as output from the buffer, channel agent *out* has also to delete the head element from the queue. This is achieved by adding an update to *out*.SEND. In addition this operation should only be performed when the queue is not empty, so that we also add an additional guard to *out*.SEND to check this property.

$$\begin{aligned} \text{FIFOBUFFEROUT} &= \{\text{BUFSEND}, \textit{out}.RECEIVE\} \text{ where} \\ \text{BUFSEND} &= \textbf{addGuard} \textit{queue} \neq \text{empty} \text{ to} \\ &\quad \textit{out}.SEND(\textit{head}(q)) \textbf{addUpd DELETE}(\textit{head}(\textit{queue}), \textit{queue}) \end{aligned}$$

These two ASMs may operate asynchronously, forming an asynchronous ASM FIFOBUFFER whose components share the common data structure *queue*. They could also be combined by interleaving, as is done in [34, Ch.3]), resulting in the following interleaved FIFOBUFFER version.

$$\text{FIFOBUFFER} = \text{FIFOBUFFERIN} \textbf{or} \text{ FIFOBUFFEROUT}$$

The operator **or** for the non-deterministic choice among rules *R*(*i*) is defined as follows<sup>4</sup>:

$$R(0) \textbf{ or } \dots \textbf{ or } R(n-1) = \textbf{choose } i < n \textbf{ do } R(i)$$

The fourth and last step of our illustration of stepwise building a machine out of components consists in refining the unbounded buffer into a bounded one with queue of maximal length  $N$ . It suffices to substitute for the rule BUFRcv its refinement by the additional guard  $\text{length}(\text{queue}) < N$ , thus preventing its application when the buffer is full.

$\text{BOUNDED FIFO BUFFER}(N) =$   
 $\text{FIFO BUFFER}(\text{addGuard } \text{length}(\text{queue}) < N \text{ to BUFRcv/BUFRcv})$

## 4. Conclusion

We have illustrated how the ASM method can help to bridge the gap between specification and design by rigorous high-level (hw/sw co-) modeling which is linked seamlessly to executable code, in a way the practitioner can verify and validate. We have explained how based upon the clear notions of ASM state and state transition and their refinements, system verification and validation at different levels of abstraction and rigor can be combined by the ASM method in a uniform way. We have defined some ASM composition operators which allow one to combine, in design and analysis, the global system view with the local view of the components. In particular this allows one to verify overall system features by theorem proving methods which exploit the model-checked properties of the components.

## Notes

1. Some of the material (in particular slides for lectures on the themes treated in the book) can also be downloaded from the AsmBook website at <http://www.di.unipi.it/AsmBook/>.
2. The C-based *XASM* system [1] has been used in [2],[3],[4] to support a uniform machine-executable ASM-specification of the static and dynamic semantics of programming languages, and similarly in an architecture and compiler co-generation project [48].
3. This is analogous to the situation in *TLA<sup>+</sup>* [34] where a specification is given by a formula in which components are represented by subexpressions.
4. For further discussion of how to combine process algebra constructs with ASMs see [8].

## References

- [1] M. Anlauff and P. Kutter. Xasm Open Source. Web pages at <http://www.xasm.org/>, 2001.
- [2] M. Anlauff, P. Kutter, and A. Pierantonio. Montages/Gem-Mex: a meta visual programming generator. TIK-Report 35, ETH Zürich, Switzerland, 1998.
- [3] M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjoerner and M. Broy, editors, *Proc.*

- Perspectives of System Informatics (PSI'99)*, volume 1755 of *Lecture Notes in Computer Science*, pages 40–53. Springer-Verlag, 1999.
- [4] M. Anlauff, P. Kutter, A. Pierantonio, and A. Sünbül. Using domain-specific languages for the realization of component composition. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 112–126, 2000.
  - [5] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, 2002.
  - [6] M. Barnett and W. Schulte. Contracts, components and their runtime verification on the .NET platform. *J. Systems and Software*, Special Issue on Component-Based Software Engineering, 2002, to appear.
  - [7] C. Beierle, E. Börger, I. Durdanović, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in *LNCS*, pages 62–78. Springer, 1996.
  - [8] T. Bolognesi and E. Börger. Abstract State Processes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003—Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 22–32. Springer-Verlag, 2003.
  - [9] E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, pages 1–43. Springer-Verlag, 1999.
  - [10] E. Börger. The ASM ground model method as a foundation of requirements engineering. In N. Dershowitz, editor, *Manna Symposium*, volume 2772 of *LNCS*. Springer-Verlag, 2003.
  - [11] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14, 2003.
  - [12] E. Börger. Teaching asms to practice-oriented students. In *Teaching Formal Methods Workshop*, pages 5–12. Oxford Brookes University, 2003.

- [13] E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2003, to appear.
- [14] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
- [15] E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A complete formal definition of the semantics of C#. Technical report, In preparation, 2003.
- [16] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.
- [17] E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.
- [18] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 361–366. Springer-Verlag, 2000.
- [19] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by Abstract State Machines: The light control case study. *J. Universal Computer Science*, 6(7):597–620, 2000.
- [20] E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 2, pages 20–90. North-Holland, 1995.
- [21] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.
- [22] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
- [23] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [24] G. Del Castillo. *The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. PhD thesis, Universität Paderborn, Germany, 2001.

- [25] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2000.
- [26] A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
- [27] A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.
- [28] C# Language Specification. Standard ECMA-334, 2001. <http://www.ecma-international.org/publications/standards/ECMA-334.HTM>.
- [29] Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at <http://research.microsoft.com/foundations/AsmL/>, 2001.
- [30] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
- [31] U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
- [32] R. Groenboom and G. R. Renardel de Lavalette. A formalization of evolving algebras. In S. Fischer and M. Trautwein, editors, *Proc. Accolade 95*, pages 17–28. Dutch Research School in Logic, ISBN 90-74795-31-5, 1995.
- [33] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley, 2003.
- [34] L. Lamport. *Specifying Systems*. Addison-Wesley, 2003.
- [35] L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer-Verlag, 2000.
- [36] W. Mueller, R. Dömer, and A. Gerstlauer. The formal execution semantics of SpecC. Technical Report TR ICS 01-59, Center for Embedded Computer Systems at the University of California at Irvine, 2001.

- [37] S. Nanchen and R. F. Stärk. A security logic for Abstract State Machines. In *TR 423 CS Dept ETH Zürich*, 2003.
- [38] A. Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 434–439, Elsevier, Amsterdam, 1994.
- [39] G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
- [40] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
- [41] J. Schmid. Executing ASM specifications with AsmGofer. Web pages at <http://www.tydo.de/AsmGofer>.
- [42] J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.
- [43] A. Schönegge. Extending dynamic logic for reasoning about evolving algebras. Technical Report 49/95, Universität Karlsruhe, Fakultät für Informatik, Germany, 1995.
- [44] R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. Technical report, Computer Science Department, ETH Zürich, 2003.
- [45] R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.
- [46] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
- [47] A. Sünbül. *Architectural Design of Evolutionary Software Systems in Continuous Software Engineering*. PhD thesis, TU Berlin, Germany, 2001.
- [48] J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for ASIPs. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pages 26–33, San Jose, CA, USA, November 2000. ACM Press.
- [49] K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.
- [50] K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, Germany, 2001.

*This page intentionally left blank*

## Chapter 17

# A NEW TIME EXTENSION TO $\pi$ -CALCULUS BASED ON TIME CONSUMING TRANSITION SEMANTICS

Marco Fischer<sup>1,2</sup>, Stefan Förster<sup>1,2</sup>, André Windisch<sup>2</sup>,  
Dieter Monjau<sup>1</sup>, Burkhard Balser<sup>2</sup>

<sup>1</sup>*Chemnitz University of Technology,  
Department of Computer Science, 09107 Chemnitz, Germany*

<sup>2</sup>*EADS Military Aircraft, New Avionics Structures,  
81663 Munich, Germany  
marco.fischer@m.eads.net*

**Abstract** This paper presents a dense time extension of the  $\pi$ -calculus based on time consuming transition semantics. The proposed extension tries to minimise the changes to the standard calculus whilst gaining enough expressiveness to model real-time systems. Syntactic rules and the operational semantics of the timed  $\pi$ -calculus are given. However, the main contribution consists of systems of equations recursively defined over timed labelled transition systems (TLTS), whose solutions provide time sets stating when processes and transitions are active.

**Keywords:** real-time systems, process algebra, time extension

### 1. Introduction

Driven by the stringent economic requirement of continuous life cycle cost reduction for modern aircraft such as the A380 or the Eurofighter, the design of avionics systems is currently making a transition from classic "box-based" design towards the design of Integrated Modular Avionics (IMA) systems. In contrast to classic avionics systems, IMA-systems exhibit a well-defined application-independent hardware (HW)/

software (SW) interface which facilitates the use of commercial-off-the-shelf (COTS) HW components and enables reuse of SW modules.

These IMA-inherent properties necessitate a shift in the accompanying design methodology towards support for HW-independent development of application software which can be freely distributed across the set of available HW resources. Clearly, this amendment of the overall design process requires extensive tool support for HW/SW system specification, refinement, and verification all of which should be founded on a unifying and sound formal foundation. One formalism capable of capturing all of these design aspects is the  $\pi$ -calculus process algebra which allows for a semantic unification of established formal specification and verification approaches.

The Automated Generation of IMA-Systems (AGS) project, currently carried out at European Aeronautic Defence and Space (EADS) Military Aircraft, aims for the automation of certain IMA system design steps. Currently, the AGS formalism [3] is based on pure  $\pi$ -calculus and thus only supports system property analysis on a causal level. This paper presents first results on the integration of time consuming transitions into the  $\pi$ -calculus underlying the AGS formalism.

The remainder of this paper is structured as follows: The next section gives an overview of related research, followed by a short introduction to  $\pi$ -calculus. In section 17.4 a time extension of  $\pi$ -calculus is proposed. Subsequently, in section 17.5 the timing information contained within the induced timed labelled transition system is used to calculate at which times a system is in some given state. The last section summarises the paper including a brief outlook at future work.

## 2. Related Work

In literature of the past two decades, a lot of different approaches to extend process algebras, such as CSP [5] or CCS [8], with a notion of time can be found. Usually this is done by introducing some additional syntax and defining its meaning through a set of transition rules. The work presented in this paper follows a similar approach. Probably the most closely related publications to name are [1, 4]. The authors extend CCS, the origin of  $\pi$ -calculus, with time. Both approaches use a two-phase execution model. In this model, processes urgently execute all possible reactions. Thereafter, all components synchronously delay for some time, before entering the next "reaction" phase. In PRTCCS the focus is on performance analysis. The proposed extension allows specification of random delays distributed according to an approximated random process. Another time extension to  $\pi$ -calculus is presented in [7].

In this paper, time is modelled by specific time events which represent the progress of one time unit. Consequently, the time domain is discrete. Timing information is captured by a timeout operator with semantics defined operationally. In [1] the primitive for capturing time information is the  $WAIT d$  operator whose semantics are given in denotational form, as this is typically done in CSP. However, in [10] operational semantics are also given for TCSP. Other approaches to time extension [6, 2] use enabling time intervals to express when actions of a process can engage with another process. This is in contrast to our proposed timing extension, where actions are enabled on a state-by-state basis and cannot be deferred in the same state.

In addition to the time extension to  $\pi$ -calculus, a novel treatment of the timing information contained within a timed labelled transition system is introduced in this paper. It makes use of systems of equations on sets, recursively defined over timed labelled transition systems. The solutions to these equation systems are sets of times at which states and transitions of a given process are active.

### 3. Brief introduction to $\pi$ -calculus

Since this paper presents an extension of  $\pi$ -calculus, a short overview over its concepts is given in this section. The  $\pi$ -calculus is based upon the notion of names, uppercase names denote *processes* and lowercase names denote *links*. The set  $P^\pi$  of  $\pi$ -calculus process expressions is defined by the syntax

$$P ::= \sum_{i \in I} \pi_i.P_i \mid P_1|P_2 \mid (new\ a)P \mid !P$$

where  $I$  is any finite indexing set and  $\pi$  denotes an action prefix. The latter are a generalisation of process actions and are defined by the syntax

$$\pi ::= x(\vec{y}) \mid \bar{x}\langle\vec{y}\rangle \mid \tau$$

where  $x(\vec{y})$  means receiving of  $\vec{y}$  along link  $x$ ,  $\bar{x}\langle\vec{y}\rangle$  represents sending of  $\vec{y}$  along link  $x$ , and  $\tau$  denotes the unobservable action - also called *silent transition*.

In the above syntactic definitions the process expression  $\sum_{i \in I} \pi_i.P_i$  is called a *summation* and 0 represents the empty sum. Each term in a summation specifies a possible continuation behaviour  $P_i$  which is *guarded* by  $\pi_i$ , the action  $\pi_i$  must occur before  $P_i$  becomes active. Thus, the ordering of terms in a summation is insignificant since selection of a particular term is determined by the occurrence of a specific action

$\pi_i$ . The process expression  $P_1|P_2$  is called *composition* and denotes the concurrent execution of  $P_1$  and  $P_2$ . The definition of a new link  $a$  within the scope of process  $P$  is specified by the *restriction*  $(\text{new } a)P$ . Finally, the process expression  $!P$  denotes a *replicator* which creates a new copy of process  $P$  each time it is accessed.

The  $\pi$ -calculus comprises a set of *reaction* and *transition rules* which rigorously formalise the above given informal description of process semantics. For a detailed formal description of these rules for the  $\pi$ -calculus the interested reader is referred to [9].

#### 4. Time Consuming Transitions

The first part of this section introduces a syntactic extension of  $\pi$ -calculus, which allows capturing of information related to time within a process expression. The second part concentrates on formalising the meaning of the resulting timed process expressions by giving operational semantics in the form of time evolution rules.

Process algebras are suitable to model a class of systems that can be described by a set of discrete states. These systems are assumed to be in a certain state at a certain time. As long as no internal reactions or interactions with the system's environment are possible, the system remains in its current state. If the system engages in a reaction or interaction it may change its state. Physically, the state of a real system is always related to information stored within the system. Thus, any transition from one state to another implies manipulation of information in some way. A natural assumption is that manipulation of any information whatsoever, consumes an amount of time greater than zero. In order to model this assumption, we try to adopt the concept of time consuming transitions to the  $\pi$ -calculus process algebra.

Table 17.1 gives an overview of the syntactic elements of timed  $\pi$ -calculus. In this EBNF the large vertical bars indicate syntactic alternatives, whereas the small vertical bar represents the parallel composition operator. The only production in addition to those of un-timed  $\pi$ -calculus - the so called TIME GUARD - allows specification of an interval  $\Delta$ , denoting a subset of the time domain  $\mathbb{R}$ . This interval may precede any SUMMATION. The symbol  $\Delta$  is a placeholder for usual interval notations. Since the given definition is inductive, time intervals may precede any process expression. Examples of valid timed process expressions are:  $[1.2; 2.5]0$ ,  $\bar{a}\langle x \rangle.[0; 5]0$  and  $(\bar{a}.[1; 3]0 \mid a.[2; 5]0)$ . Notably, the syntax disallows summation terms to begin with a time guard<sup>1</sup>, which turns out to be important when defining the operational semantics of the calculus.

Table 17.1. Timed  $\pi$ -calculus syntax.

TIMED $\pi$ -PROCESS:	$P ::= P^\Delta \mid P^\Sigma \mid P^! \mid P^0$
ACTION PREFIX:	$P^{act} ::= \pi.P$
	$\pi ::= \alpha(\vec{\alpha}) \mid \bar{\alpha}\langle\vec{\alpha}\rangle \mid \tau$
SUMMATION:	$P^\Sigma ::= P^{act} (+ P^\Sigma)^*$
COMPOSITION:	$P^! ::= P ( \mid P^! )^*$
REPLICATION:	$P^! ::= !P^\Sigma$
RESTRICTION:	$P^\nu ::= \text{new } \vec{x} P$
INACTION:	$P^0 ::= 0$
TIME GUARD:	$P^\Delta ::= \Delta P$

Intuitively, the meaning of the proposed syntactic extension can be described as follows. Anytime a process engages in some action  $\alpha$ , it evolves into the continuation process specified after the action prefix<sup>2</sup>. In the timed calculus, the continuation can be a process of the form  $\Delta P$ , expressing that the state represented by  $P$  has not been reached yet. More precisely, the lower limit of  $\Delta$ , denoted  $\inf_\Delta$ , is the amount of time which must pass before  $P$  can be reached. That is, any delay  $d < \inf_\Delta$  implies  $P$  is not reached. After the passage of  $\inf_\Delta < d \leq \sup_\Delta$ , the question whether  $P$  is reached or not is open. Both answers are equally valid. Only an advancing of time beyond the upper limit of  $\Delta$  implies that  $P$  is reached. In other words,  $\Delta$  specifies the range of durations that a transition between states may have.

A diagram illustrating a timed state transition of a simple process expression is shown in figure 17.1. The process expression specifies a time guard  $[t_{min}; t_{max}]$  and a continuation process  $P$ . The ordinate of the diagram indicates time progress and the abscissa depicts the shift from source state to the target state. Both axes represent dense domains. In particular, the state axis can be thought of as a continuum

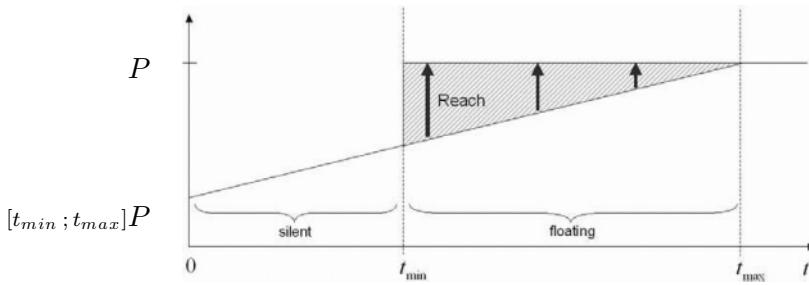


Figure 17.1. Time consuming transition.

of intermediate states, each of which is time guarded by a different interval. The time axis measures time relative to the occurrence of the preceding action that has lead to the process expression in discussion. A time transition can be divided into two phases as defined below. As time passes, the time guard decreases such that the intermediate states approach  $P$ . During the floating phase,  $P$  can be reached immediately as indicated by the vertical arrows.

**DEFINITION 17.1** *Given a time guarded process  $\Delta P$ , the interval  $[0; \inf_\Delta)$  is called the silent phase. During this phase there are neither interactions with other processes, nor internal reactions among components of  $P$ . The predicate silent on process expressions is defined as*

$$\text{silent } P = \begin{cases} \text{true} & : \forall x \in \Delta : x > 0 \\ \text{false} & : \text{otherwise} \end{cases}$$

*A silent process can do nothing except letting time pass.*

**DEFINITION 17.2** *Given a time guarded process  $\Delta P$ , the floating phase coincides with the interval  $\Delta$ . The predicate floating on process expressions is defined as*

$$\text{floating } P = \begin{cases} \text{true} & : 0 \in \Delta \\ \text{false} & : \text{otherwise} \end{cases}$$

*At any point during the floating phase there exist two possibilities for further evolution. Firstly, the process continues to be floating and allows time to pass. Alternatively, the target state may be reached immediately. Note that for processes with no time guard,  $\Delta$  is regarded to be the empty set.*

Another predicate is that of process stability. In un-timed algebras a process is usually defined as stable if it has no internal transitions, i.e.

if  $\neg(P \xrightarrow{\tau})$ . This predicate is extended to timed stability in the timed algebra as follows.

**DEFINITION 17.3** *A process expression  $P$  is said to be time-stable, if and only if*

$$\neg\text{silent } P \wedge \neg\text{floating } P \wedge \neg(P \xrightarrow{\tau}),$$

i.e. if  $P$  is stable, but neither silent nor floating. Note that  $P$  may offer actions to its environment, while still being time-stable.

The semantics of time guards shall be defined in the form of a labelled transition relation  $\xrightarrow{d} \in \mathcal{P} \times \mathbb{R}^+ \times \mathcal{P}$ , called *time evolution*. Symbols  $\mathcal{P}$  and  $\mathbb{R}^+$  denote the sets of process expressions and real numbers, respectively. Each element of a time evolution is a transition labelled with a duration  $d \in \mathbb{T}$  and is referred to as an *evolution transition*. There are five transition rules that characterise time evolution. The transition rule PROG introduces the possibility of time progress for time guarded processes. Accordingly,  $\Delta P$  can wait for  $d$ , thereby evolve into  $\Delta' P$ , where the values of  $\Delta'$  are those of  $\Delta$  decreased by  $d$ . Valid values of  $d$  range over the interval  $(0; \sup_{\Delta}]$ . PROG<sub>Stable</sub> is a rule concerning time-stable processes. It implies that any time-stable process has the capability to let time pass arbitrarily. REACH accounts for the fact that any floating process may reach its target state immediately. Hence, the inferred transition is labelled with  $d = 0$ . PROG<sub>Par</sub> allows the composition of two processes to delay, if both have the capability to delay the same amount of time and if the composition does not yield internal transitions. This ensures synchronous time progress within concurrent components of a system. Finally PROG<sub>Res</sub> expresses that restriction does not affect a process's capability to delay. The transition rule for structural congruence of Milner is also applicable for evolution transitions, although not explicitly given as an evolution rule here. A summary of the transition rules explained above is presented in table 17.2.

Having the operational semantics for timed process expressions at hand, it is possible to define a timed labelled transition system induced by the transition rules. The resulting TLTS has action transitions as well as evolution transitions.

**DEFINITION 17.4** *The timed labelled transition system of a timed process expression  $P$  is a tuple  $(L, T, S)$ , where*

$L = \text{Act} \cup \mathbb{R}^+$  is the set of labels,

$T \subset (\mathcal{P} \times L \times \mathcal{P})$  is the set of exactly those transitions which can be inferred from  $P$  by the transition rules,

$$\text{PROG: } \Delta P \xrightarrow{d} (\Delta - d)P \quad d \in (0; \sup_{\Delta})$$

$$\text{PROG}_{Stable}: \frac{\text{time-stable } P}{P \xrightarrow{d} P} \quad d \in \mathbb{R}^+$$

$$\text{REACH: } \frac{\text{floating } P}{\Delta P \xrightarrow{0} P}$$

$$\text{PROG}_{Par}: \frac{P \xrightarrow{d} P' \quad Q \xrightarrow{d} Q'}{P \mid Q \xrightarrow{d} P' \mid Q'} \quad \neg(P \mid Q \xrightarrow{\tau})$$

$$\text{PROG}_{Res}: \frac{P \xrightarrow{d} P'}{\text{new } \vec{x} P \xrightarrow{d} \text{new } \vec{x} P'}$$

Table 17.2. Time evolution rules.

$\mathcal{S} \subset \mathcal{P}$  is the set of all states involved in any transition in  $T$ .

The alphabet consists of action labels  $Act$  as usual in  $\pi$ -calculus and positive real numbers including zero  $\mathbb{R}^+$ .

In figure 17.2 the TLTS of process expression  $P \stackrel{\text{def}}{=} \Delta_0 a.P$  is shown. Each state is represented by a rounded box containing the corresponding process expression. Symbols  $\Delta_i$  denote time guards of silent processes, whereas  $\Lambda_i$  denote time guards of floating processes. For any action transition  $\xrightarrow{\alpha}$  a solid arrow labelled  $\alpha$  is drawn between source and target state. Evolution transitions  $\xrightarrow{d}$  are depicted by dotted arrows. Their label is either a real number equal to the transition duration  $d$ , or it is an interval denoting a range of possible durations. Hence, transitions labelled by some interval  $\Delta$  supersede an infinite collection of transitions, each labelled by a distinct number  $d \in \Delta$ . As can be seen in figure 17.2, even a simple process expression yields an infinite TLTS. However, the states in the shaded area can be regarded as one super-state capturing the possible behaviour. That is, all transitions to and from this state are known independently from their context. Starting from process expression  $P$ , there are two evolution transitions. The transition  $\xrightarrow{(0; \inf \Delta_0)}$  leads to subsequent silent processes. As further time passes, all evolutions finally lead to  $\Lambda_0 a.P$ , where the process floats for the first time. As stated in definition 17.2 this process has two possibilities: (1) reach the target state or (2) consume time. Exactly this is expressed by the evolution transitions  $\xrightarrow{0}$  and  $\xrightarrow{(0; \sup \Lambda_0)}$ . Following the latter, again a super-state of

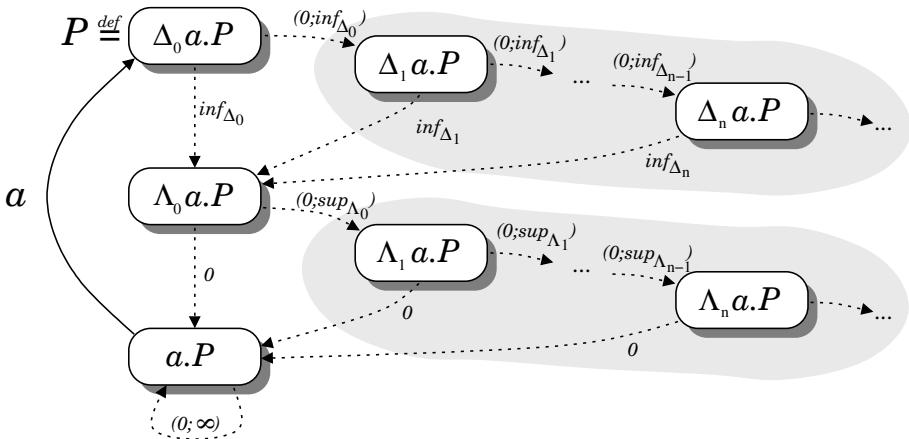


Figure 17.2. TLTS of a time guarded process.

floating processes is entered as indicated by the lower shaded area. All outgoing transitions  $\xrightarrow{0}$  finally lead to  $a.P$ . The evolution transition from  $a.P$  can be inferred by  $\text{PROG}_{\text{Stable}}$ , since  $a.P$  is a stable process (cf. definition 17.3).

## 5. Temporal properties of TLTS

A fundamental question about real-time systems is what the state of a system will be at a certain time and which actions can or cannot be performed at that time. The answer to such questions can be computed from the TLTS of a system as defined by definition 17.4. Given the state  $S \in \mathcal{S}$  of some system, let **ACTIVE** be a set of times, such that at each  $t \in \mathbf{ACTIVE}$  the current state of the system is  $S$ . The sets **ACTIVE** and **ARRIVE** are defined as the solution to a system of equations between functions over sets as follows.

**DEFINITION 17.5** *The function  $\text{ACTIVE}(\cdot) : \mathcal{S} \rightarrow P(\mathbb{R}^+)$  determines the set of times at which a system is in the state  $S \in \mathcal{S}$ :*

$$\text{ACTIVE}(S) = \bigcup_{t_I \in T_I(S)} \text{ARRIVE}(t_I),$$

where  $T_I(S) = \{(Q, \alpha, Q') \in \mathcal{S} \times L \times \mathcal{S} \mid Q' = S\}$

is the set of incoming transitions of state  $S$ .

**DEFINITION 17.6** *The function  $ARRIVE(\cdot) : \mathcal{S} \times L \times \mathcal{S} \rightarrow P(\mathbb{R}^+)$  determines the set of times at which the target state of a transition can be reached:*

$$ARRIVE((S, \alpha, S')) = ACTIVE(S) \oplus DURATION((S, \alpha, S'))$$

where

$$DURATION((S, \alpha, S')) = \begin{cases} 0 & : \alpha \in Act \\ \alpha & : \alpha \in \mathbb{R}^+ \end{cases}$$

is the duration of a transition according to its label.

Note that the  $\oplus$  operation in definition 17.6 is defined as  $A \oplus d = \{x \mid x = a + d \quad \forall a \in A\}$ , i.e. the set of all members of  $A$  increased by  $d$ . The application of  $ACTIVE(\cdot)$  to some state of a TLTS gives a system of equations whose solution is the desired set **ACTIVE**. Doing so for the TLTS in figure 17.2 yields the times of the states and transitions<sup>3</sup>. For notational purposes names instead of process expressions are used to refer to the states:

$$P_0 \stackrel{\text{def}}{=} \Delta_0 a.P \quad P_1 \stackrel{\text{def}}{=} \Lambda_0 a.P \quad P_2 \stackrel{\text{def}}{=} a.P$$

The resulting system of equations is

$$ACTIVE(P_0) = ACTIVE(P_2) \oplus \underbrace{DURATION((P_2, a, P_0))}_{=0} \quad (17.1)$$

$$ACTIVE(P_1) = ACTIVE(P_0) \oplus \underbrace{DURATION((P_0, \inf_{\Delta_0}, P_1))}_{=\inf_{\Delta_0}} \quad (17.2)$$

$$\begin{aligned} ACTIVE(P_2) &= ACTIVE(P_1) \oplus \underbrace{DURATION((P_0, (0; \sup_{\Delta_0}), P_1))}_{=(0; \sup_{\Delta_0})} \\ &\cup ACTIVE(P_2) \oplus \underbrace{DURATION((P_2, (0; \infty), P_2))}_{=(0; \infty)} \end{aligned} \quad (17.3)$$

Eliminating the recursion in  $ACTIVE(P_2)$  yields

$$\begin{aligned} ACTIVE(P_2) &= \bigcup_{n \in \mathbb{N}^+} \left( ACTIVE(P_1) \oplus (0; \sup_{\Lambda_0}) \right) \oplus \underbrace{n \cdot (0; \infty)}_{=(0; \infty)} \\ &= \left( ACTIVE(P_1) \oplus (0; \sup_{\Lambda_0}) \right) \oplus (0; \infty) \end{aligned} \quad (17.4)$$

By substituting (17.4) in (17.1), it can be shown that the set of times at which  $P_0$  is active is given by

$$\begin{aligned} ACTIVE(P_0) &= \left( ACTIVE(P_1) \oplus (0; \sup_{\Lambda_0}) \right) \oplus (0; \infty) \\ &= \left( \left( ACTIVE(P_0) \oplus \inf_{\Delta_0} \right) \oplus (0; \sup_{\Lambda_0}) \right) \oplus (0; \infty) \\ &= ACTIVE(P_0) \oplus [\inf_{\Delta_0}; \infty) \end{aligned} \quad (17.5)$$

This result expresses that  $P_0$  is active at the earliest  $\inf_{\Delta_0}$  after  $P_0$  was active and any time after that, which is exactly what the TLTS in figure 17.2 shows intuitively. It takes at least  $\inf_{\Delta_0}$  to get from  $P_0$  to  $P_2$  and back to  $P_0$ , but  $P_2$  has the capability to delay an arbitrary amount of time, such that  $P_0$  can be reached again any time later. Figure 17.3(a) gives a graphical representation of  $ACTIVE(P_0)$ . An interesting case would be if  $P_0$  was in a cycle with duration from the interval  $[a; b]$  instead of  $[\inf_{\Delta_0}; \infty)$ , i.e.:

$$ACTIVE(P_0) = ACTIVE(P_0) \oplus [a; b] \quad (17.6)$$

Then, all times greater than a certain  $\epsilon$  were contained in  $ACTIVE(P_0)$ . Consequently,  $P_0$  could be active at anytime after  $\epsilon$  as depicted in figure 17.3(b). In this case, the value of  $\epsilon$  would be the least common multiple of  $a$  and  $b$ .

## 6. Conclusion and Future Work

This paper presented a time extension of  $\pi$ -calculus, based on time consuming transitions. Time evolution rules together with the standard transition and reaction rules induce a timed labelled transition system. Timing information is captured by real-number labels representing time progress. The definition of time sets **ACTIVE** and **ARRIVE** as solutions to equation systems over TLTS constitute a new way of gaining

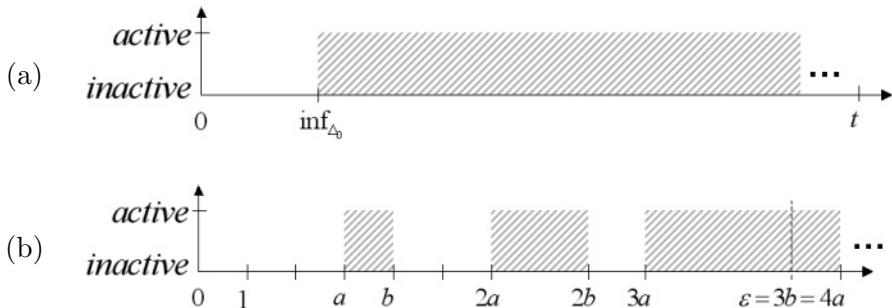


Figure 17.3. Illustration of  $ACTIVE(P_0)$  according to equation (a) 17.5 and (b) 17.6.

characteristic timing information of timed processes. Future work comprises development of tool support as well as integration into the AGS methodology. Furthermore, general properties of the proposed equation systems have to be analysed.

## Notes

1. For example  $(a + [1; 2]b)$  is not a valid process expression.
2. This is a part of the usual transition rules.
3. Actually, the derivation uses a simplified, but equivalent TLTS with only three states  $P_{0,1,2}$  and the transition between  $P_1$  and  $P_2$  being  $P_1 \xrightarrow{0; \sup_{\Delta_0}} P_2$ .

## References

- [1] Davies, J. (1993). Specification and Proof in Real-Time CSP. Technical report, Oxford University.
- [2] Fidge, Colin and Žic, John (1995). An expressive real-time ccs. Technical report, Software Verification Research Centre, University of Queensland.
- [3] Förster, S., Windisch, A., Fischer, M., Monjau, D., and Balser, B. (2003). Process Algebraic Specification, Refinement, and Verification of Embedded Systems. In *Proceedings of FDL03*.
- [4] Hansson, Hans and Jonsson, Bengt (1990). A Calculus for Communicating Systems with Time and Probabilities. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida. IEEE Computer Society Press.
- [5] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

- [6] Leduc, G. (1992). An upward Compatible Timed Extension to LOTOS. In Parker, K. and Rose, G., editors, *Formal Description Techniques*, volume IV, pages 217–232, North-Holland, Amsterdam.
- [7] Lee, Jeremy Y. and Žic, John (2002). On modeling real-time mobile processes. In Oudshoorn, Michael, editor, *Proceedings of 25th Australasian Computer Science Conference 2002 (ACSC2002)*, volume 4, Melbourne, Australia.
- [8] Milner, Robin (1989). *Communication and Concurrency*. Prentice Hall.
- [9] Milner, Robin (1999). *Communicating and Mobile Systems: The  $\pi$ -Calculus*. Cambridge University Press. ISBN: 0-52164320-1.
- [10] Schneider, Steve (1995). An operational semantics for timed CSP. *Information and Computation*, (116(2)):193–213.

*This page intentionally left blank*

## Chapter 18

# MODELING CHP DESCRIPTIONS IN LABELED TRANSITIONS SYSTEMS FOR AN EFFICIENT FORMAL VALIDATION OF ASYNCHRONOUS CIRCUIT SPECIFICATIONS

Menouer Boubeker<sup>1</sup>, Dominique Borrione<sup>1</sup>, Laurent Mounier<sup>2</sup>, Antoine Sironi<sup>1</sup>, Marc Renaudin<sup>1</sup>

<sup>1</sup>*TIMA Laboratory, 46 Avenue Felix Viallet, 38100 Grenoble, France*

<sup>2</sup>*VERIMAG Centre Equation, 2 avenue de Vignate, 38610 Gieres, France*

**Abstract** This work addresses the analysis and validation of CHP specifications for asynchronous circuits, using property verification tools. CHP semantics, initially given in terms of Petri Nets, are reformulated as labeled transition systems. Circuit specifications are translated into an intermediate format (IF) based on communicating extended finite state machines. They are then validated using the IF environment, which provides model checking and bi-simulation tools. The direct translation must be optimized to delay state explosion. Performance studies are reported.

## 1. Introduction

With the integration of several complex systems on only one chip, synchronous logic design encounters major problems (distribution of clock, energy, modularity). Asynchronous circuits show interesting potentials in several fields such as the design of microprocessors, smart cards and circuits with low power consumption[1]. The design of effective asynchronous circuits in terms of performance and correctness requires rigorous design and validation methods.

Typically, asynchronous circuits are specified at logic level, using a CCS or a CSP-like formalism[1]. Indeed, an asynchronous circuit can

be seen as a set of communicating processes, which read a piece of data starting from input ports, carry out a treatment and finally write on output ports. In our work, asynchronous circuit specifications are written in an enriched version of the CHP (Communicating Hardware Processes) language initially developed by Alain Martin in Caltech[2].

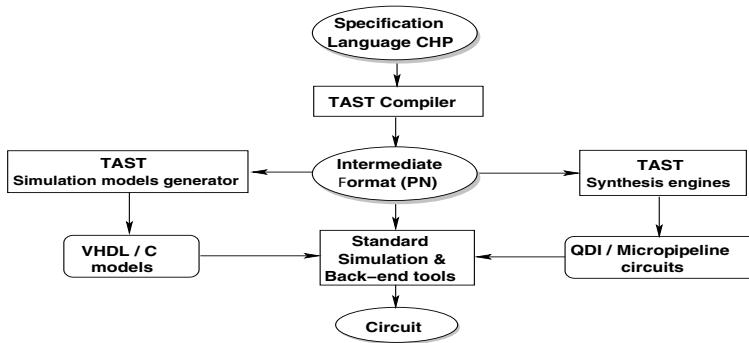


Figure 18.1. The TAST design flow

The TAST design flow is shown in the square boxes of Figure 18.1 [3, 4]. CHP specifications are interpreted in terms of Petri Nets. Process decomposition and refinements are performed on the Petri Net formalization, from which a choice of architectural targets are available: micro-pipeline, quasi delay insensitive circuit, or synchronous circuit. Dedicated synthesis engines produce structural gate networks, in source VHDL, for simulation and back-end processing using commercial CAD tools. On the other hand, behavioral VHDL and C models can be generated for validation by simulation. This approach has supported, up to silicon fabrication, the design of two asynchronous microprocessors: ASPRO [5] and MICA[6].

The goal of our work is to introduce formal methods into the asynchronous synthesis flow. A first feasibility study was performed on an industrial symbolic model checker that is routinely used to perform property checking on RTL designs[7]. The approach consisted in translating the Petri Net, interpreted as a finite state machine, as a pseudo synchronous VHDL description, where the only purpose of the pseudo clock thus introduced was to make the result of each computation cycle a visible state. In essence, our translation performed a static pre-order reduction, by replacing all independent event interleavings by data-flow concurrency. Due to the semantics of process communication by signals at RT level, abstract communication channels had to be synthesized

prior to property verification. This approach gives good results on models after communication expansions, i.e. after essential design decisions have been taken, among which data encoding and communication protocol. But this is too late in the design process to validate the initial specifications.

Our second approach consists in using formalisms and tools coming from the field of software validation, in particular distributed systems, whose model of execution is similar to the asynchronous circuits one. CHP specifications are analyzed and translated in terms of LTS (labeled transitions systems) with guarded commands. These LTS descriptions explicitly integrate channels and simple read and write actions. They appear to be more appropriate for the formal validation of initial CHP specifications. Our work being devoted to specification semantics

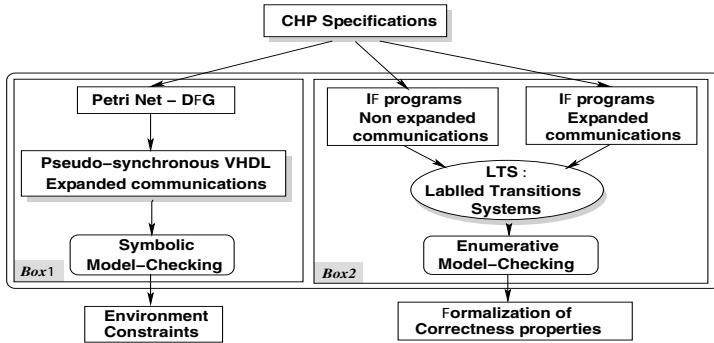


Figure 18.2. Formal validation flow.

and verification methods rather than to the development of one more model-checker, we again looked for available existing software. We selected IF[8, 9], an intermediate format developed in Verimag laboratory to describe asynchronous processes and LTS. The construction of the IF descriptions is carried out so as to preserve the asynchronous specifications semantics in terms of choice structures, communication and concurrency. Resulting programs are compiled towards a LTS and eventually submitted to the CADP toolset[10] for verification. To obtain a correct behavior, a set of environment assumptions must be associated to each communication channel, ensuring that each input wire behavior is correct. The environment behavior can be modeled as a concurrent IF process. In this approach, we also consider validating the specifications at lower levels, i.e. after the communications expansion according to a four phase handshake protocol. Then we can perform verification between specifications: before and after the communication expansion.

Figure 18.2 shows the formal validation flow for the two approaches. Box 1 corresponds to the pseudo-synchronous model, checked by symbolic methods. Box 2 displays the use of enumerative methods on asynchronous models. In both cases, we verify safety properties such as freedom from deadlock or livelock, and temporal behavior.

The rest of the paper is organized as follows. Section 2 reviews the essential features of CHP and its semantics both in terms of Petri Net and LTS. Section 3 reports the results of performance studies on the two approaches. In Section 4, as a case study, we present the applicability of the approach on a real-life asynchronous circuit: a four stage Filter. Section 5 ends the paper with our conclusions.

## 2. Translation from CHP to Petri Nets and IF

Numerous formalisms have been proposed to model the execution of concurrent processes. We consider here two of them, based respectively on Petri Nets and on the IF intermediate format. The operational semantics of these two models can be defined in terms of Labeled Transition Systems (LTS). Then we detail the translation schemes that have been implemented in the tool.

### 2.1 The Petri Nets and IF models

**TAST Petri Nets.** A Petri Net (PN) is a quadruple  $\langle S, T, F, M_0 \rangle$  where  $S$  and  $T$  are finite disjoint sets of places and transitions,  $F \subseteq (ST) \cap (T \times S)$  is an input/output flow relation, and  $M_0 \subseteq 2^S$  is the initial marking. The operational semantics of a PN is classically defined in terms of LTS by using markings (i.e., elements of  $2^S$ ) to represent global system states. A global transition holds between two markings  $M$  and  $M'$  if and only if there exists a set of PN transition  $\mathcal{T} \subseteq T$ , those input places are contained in  $M$ , and such that  $M'$  can be obtained from  $M$  by replacing the input places of  $\mathcal{T}$  by their output places. Thus, a PN can easily describe the control flow of a process with concurrent statements.

Data operations can be taken into account on this basic model by extending a PN with a set of variables. Basic statements (data assignments) can then be attached to places, and conditions (or guards) can be attached to transitions. In the TAST PN model, variables are typed and three basic statements are available: variable assignments and value emission or reception via an external communication port.

**The IF intermediate format.** We consider here a subset of the IF intermediate format in which a system description is a quadruple  $\langle$

$A, E, G, P >$  where  $A$  is a set of *disjoint extended automata*  $A_i$ ,  $E$  a *composition expression*,  $G$  a set of (typed) *global variables* and  $P$  a set of *communication ports*. Each extended automaton  $A_i$  is a tuple  $\langle Q_i, L_i, R_i, q_{0i}, V_i \rangle$  where  $Q_i$  is finite set of *control states*,  $L_i$  a finite set of *labels*,  $R_i \subseteq Q_i \times L_i \times Q_i$  a *transition relation*,  $q_{0i}$  the *initial state* and  $V_i$  a set of (typed) *local variables*. Labels are of the form "*gcoms*" where  $g$  is a condition (a Boolean expression over  $G \cap V_i$ ), com is a value emission (*plexpression*) or a value reception ( $p?x$ ) via a communication port  $p$  of  $P$ , and  $s$  is a sequence of variable assignments. In addition control states can be declared *unstable*, to better tune the atomicity level of transitions (transition sequences between unstable states are considered atomic).

The communication primitive is based on *rendez-vous*: a value emission (resp. reception) on a given communication port  $p$  can be executed by automaton  $A_i$  if and only if another automaton  $A_j$ , synchronized with  $A_i$  on port  $p$ , is ready to perform a corresponding value reception (resp. emission) on the same port. The set of synchronization ports between each automata pair is given in a LOTOS-like algebraic style by the composition expression  $E$ . More precisely, this expression is built upon a general binary composition operator  $A_i|[G]|A_j$ , allowing to synchronize (by rendez-vous) the execution of automata  $A_i$  and  $A_j$  on the port set  $G$ . The complete operational semantics of the IF model in terms of LTS is detailed in [9].

## 2.2 CHP components

A TAST CHP component is made of its communication interface, a declaration part and a set of concurrent CHP processes. The communication interface is a directed port list (similar to VHDL) and the declaration part lists the channels used for inter-process communications. Channels interconnect processes through point-to-point, asymmetric, memory less, message passing and compatible protocol links.

The translation schemes we propose from a CHP component to the Petri Net and IF models are the following:

**from CHP to PN:** each CHP process is translated into a single Petri Net, and inter-process communications are expanded with an explicit communication protocol based on global variables (see section 2.4 below);

**from CHP to IF:** each CHP process is translated into an IF extended automaton and the channels declaration is translated into a corresponding composition expression.

These two parts are detailed in the rest of the section.

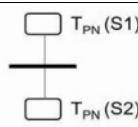
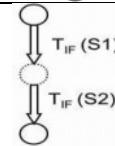
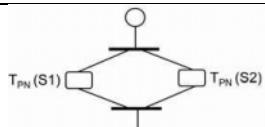
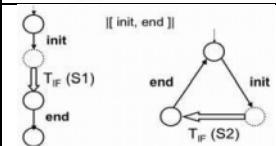
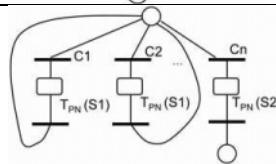
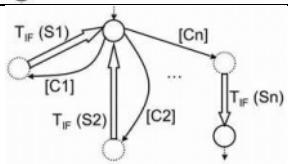
## 2.3 CHP processes

The abstract syntax of a TAST CHP process is given below:

<i>Process</i>	$::= \text{Interface Declaration}^* \text{ Statement}^*$
<i>Statement</i>	$::= \text{Basic Statement} \mid \text{Sequential_Composition} \mid \text{Parallel Composition} \mid \text{Guarded Commands}$
<i>Basic Statement</i>	$::= \text{var} := \text{expression} \mid p! \text{expression} \mid p? \text{var}$
<i>Sequential Composition</i>	$::= \text{Statement} ; \text{Statement}$
<i>Parallel Composition</i>	$::= \text{Statement}, \text{Statement}$
<i>Guarded Commands</i>	$::= @[\ ( \text{Guard} \Rightarrow \text{Statement} ; \text{Termination} )^* ]$
<i>Termination</i>	$::= \text{loop} \mid \text{break}$
<i>Guard</i>	$::= \text{expression} \mid \text{port} \#$

Roughly speaking, a process description consists in a communication interface, a declaration part and a statement part. The communication interface is a list of ports connected to local channels or component ports, and the declaration part is a list of typed local variables.

Table 18.1. Translation of CHP main statements

CHP term t	TPN (t)	TIF (t)
Basic statement S: <i>var</i> := <i>expression</i> <i>p!</i> <i>expression</i> <i>p?</i> <i>var</i>		
Sequential composition: <i>S1; S2</i>		
Parallel composition: <i>S1, S2</i>		
Guarded Commands: $@[C1 \Rightarrow S1; \text{loop}$ $C2 \Rightarrow S2; \text{loop}$ $\dots$ $Cn \Rightarrow Sn; \text{break}]$		

The statement part is inspired from A. Martin's original CHP proposal: a statement can be either a *basic statement* (variable assignment, value emission or reception through a communication port), a *sequential* or *parallel* composition, or a general *guarded commands* block. A *guard* is a Boolean expression which may include a channel *probe* (denoted with the  $\#$  operator).

The translation of the communication interface and data parts into the PN and IF models is rather immediate and we focus here on the translation of the statement part. First, all CHP basic statements have their direct counterparts in both PN and IF. Furthermore, CHP control structures (sequential and parallel composition, guarded commands) can also be easily expressed within these models using compositional "patterns". Therefore, the translation from a CHP process to PN and IF can be defined in a classical way by structural syntactic induction. Formally, we consider two translation functions,  $T_{PN}$  and  $T_{IF}$ , associating respectively a PN and IF pattern to each CHP term. These functions are (recursively) defined in Table 18.1.

The only non-straightforward case is the translation of CHP parallel statement composition into IF ( $T_{IF}(S1, S2)$ ). Since parallelism is not allowed *inside* IF extended automata, we need to introduce an auxiliary automaton for statement  $S2$ , to be synchronized with  $S1$  via two extra communication ports *init* and *end*.

## 2.4 Inter-process communications and probes

CHP inter-process communications (value emission  $p!expression$  and value reception  $p?x$ ) are based on a high-level *rendez-vous* mechanism. Their translation into IF is therefore straight forward since the corresponding primitive is available. This makes possible, at reasonable cost, the verification of a circuit design at a specification level. However, to be synthesized on a gate level, this high-level communication primitive has to be expanded into a concrete protocol. The protocol we choose here is a (classical) four-phase handshake protocol[11] based on global variables: for each channel  $P$ , a variable  $P$  stores the exchanged value, and two Boolean variables  $P_{req}$  and  $P_{ack}$  manage the handshake (in practice  $P_{req}$  and  $P$  can be encoded using a single variable). Table 18.2 shows the translation of this expansion phase into both PN and IF, assuming that the "sender" process initiates the communication (it is *active* on this channel) and that the "receiver" waits for a value (it is *passive* on this channel). On the reverse situation (active reception and passive emission) a dual protocol has to be applied.

Table 18.2. Communication expansion

Communication expansion	PN	IF
(passive) value reception $p?x$	<pre> graph TD     T1(( )) -- "P_req=1" --&gt; P1(( ))     P1 -- "x:=P; P_ack:=0" --&gt; T2(( ))     T2 -- "P_req=0" --&gt; P2(( ))     P2 -- "P_ack:=1" --&gt; T3(( ))   </pre>	<pre> graph TD     T1(( )) -- "P:=expression; P_req:=1" --&gt; P1(( ))     P1 -- "P_ack=0" --&gt; T2(( ))     T2 -- "P_req:=0" --&gt; P2(( ))     P2 -- "P_ack:=1" --&gt; T3(( ))   </pre>
(active) value emission $p!expression$	<pre> graph TD     T1(( )) -- "P:=expression; P_req:=1" --&gt; P1(( ))     P1 -- "P_ack=0" --&gt; T2(( ))     T2 -- "P_req:=0" --&gt; P2(( ))     P2 -- "P_ack:=1" --&gt; T3(( ))   </pre>	<pre> graph TD     T1(( )) -- "P:=expression; P_req:=1" --&gt; P1(( ))     P1 -- "[P_ack=0]" --&gt; T2(( ))     T2 -- "P_req:=0" --&gt; P2(( ))     P2 -- "P_ack:=1" --&gt; T3(( ))   </pre>

This communication expansion also provides a straightforward translation of the CHP *probe* operator. This Boolean operator allows a process to check on a passive channel  $P$  if another process is waiting for a communication on  $P$ . This check can then simply be done by testing the value of variable  $P_{req}$ .

## 2.5 Optimizations

The efficiency of the IF verification tools is drastically influenced by the size of the underlying LTS associated to the IF model. It is therefore crucial to keep this size as small as possible. To this purpose, the general translation scheme proposed above can be improved in some particular situations. These optimizations fall into two categories:

**1. Transition atomicity:** IF transitions can be labeled with a guard, a communication and several variable assignments. In some cases this transition structure allows to suppress intermediate control states produced by function  $T_{IF}$ , or at least to explicitly declare them as *unstable*. This increases the atomicity level of the transition, and thus reduces the size of the underlying LTS (in particular when this transition is executed in parallel with transitions of other processes). These extra states are the "dotted" ones in Table 18.1, and they can be suppressed whenever the subsequent statement  $S_i$  is a *basic* statement.

**2. Explicit interleaving:** The use of an auxiliary IF automaton to translate CHP parallel statement composition is also very "state consuming" (extra communications init and end increase the size of the underlying LTS). Whenever statements  $S_1$  and  $S_2$  are basic statements, a better solution consists in translating the parallel composition  $S_1, S_2$  with an explicit transition interleaving. Note that this interleaving can

even be removed whenever  $S_1$  and  $S_2$  are independent statements (e.g., assignments to distinct local variables). Figure 18.3 below illustrates this optimized translation of CHP statement  $S_1, S_2$  in both situations. The "dotted" state can be suppressed if  $S_2$  is a basic variable assignment. These optimizations have been integrated into the CHP to IF translation tool.

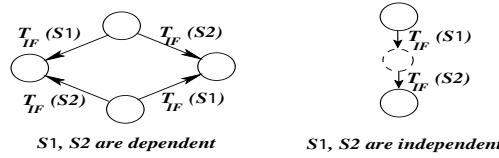


Figure 18.3. Explicit interleaving of parallel composition.

### 3. Performance study

Figure 18.4 shows the Petri-Net of a selector circuit, that we used as benchmark. The behavior is the following. Input channel  $C$  is read in a local variable  $ctrl$  which is tested using a deterministic choice structure. The value read from input channel  $E$  is propagated to output channel  $S_1$  if  $ctrl$  is 0, to  $S_2$  if  $ctrl$  is 1, and to both  $S_1$  and  $S_2$  in parallel if  $ctrl$  is 3.

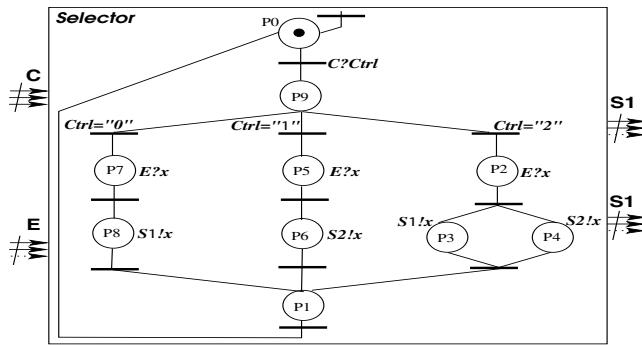


Figure 18.4. The basic selector

We have varied the bit-width of the data channels ( $E, S_1, S_2$ ), the size of control channel  $C$  and consequently the number of alternative paths (see Figure 18.5.a), and the number of concurrent paths (see Figure 18.5.b). We measured verification time, maximum memory occupancy

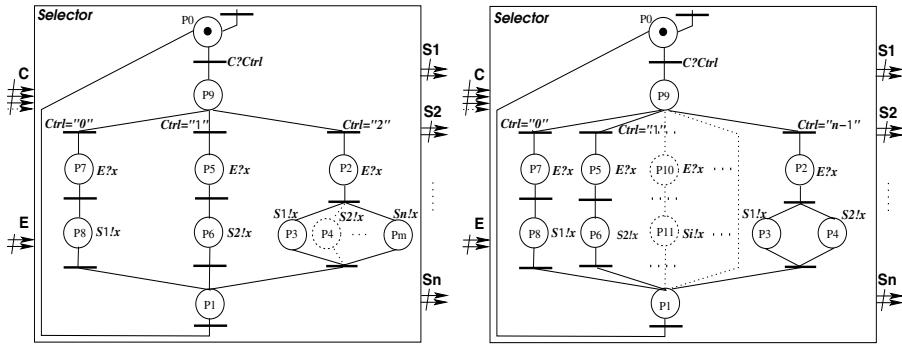


Figure 18.5. (a) Varying the control size, (b) Varying the number of concurrent paths

and BDD/LTS size on two verification systems: a commercial symbolic model checker, and the CADP tool set.

To conduct the experiment with the first system, we applied the "pseudo-synchronization" principle described in [7], and translated the Petri Net into synthesizable RTL VHDL. At this level, *rendez-vous* on a channel is not available, and communications must be expanded. In the second case, we translated the CHP description into IF, as described in section 2, and produced a model for the two cases: with and without expansion of the communications.

The experiments reported here concern a typical selector characteristic property, expressed in the appropriate property language for each tool: *If an incoming request ( $C = x"1"$ ) arrives, then a write will eventually occur on  $S1$ .* The results, computed on a SUN-Blade-100 Sparc with 640 MB memory are displayed in Tables 18.3, 18.4, 18.5. In the three Tables, for IF/CADP, we measure separately the model generation and the property verification times: the model can be stored and directly reused for another property, and the information was easy to get. For FormalCheck, the verification time includes the construction of the model. Time is expressed in the form mn:sec or mn:sec,ts (ts is tenth of a second).

Table 18.3 shows the effect of varying the bit width of the input-output data (Figure 18.4). CADP shows little sensitivity to the data width parameter when communications are not expanded. In the expanded case, both systems show a space and time explosion, the enumerative CADP performs better under 16 bits, and the symbolic FormalCheck performs better above.

Table 18.4 shows the effect of varying the size of the control input and the number of alternative execution paths (Figure 18.5.a). Once again,

Table 18.3. Varying Data size

Data size		2	4	8	16	24	32
a) Measures without expansion of communications							
<i>IF/CADP</i>	Size of LTS: trans/states	0.42e2/ 0.2e2	1.08e2/ 0.44e2	3.12e2/ 0.88e2	1.01e3/ 1.77e2	2.09e3/ 2.64e2	3.55e3/ 3.52e2
	Memory size	8.17	8.17	8.18	8.18	8.22	8.23
	generation time	00:00,5	00:00,5	00:00,5	00:00,6	00:00,6	00:00,6
	Verification time	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,2
b) Measures with expansion of communications							
<i>IF/CADP</i>	Size of LTS: trans / states	1.86e4/ 5.66e3	5.18e4/ 1.53e4	1.65e5/ 4.77e4	5.59e6/ 1.64e5	1.24e6/ 3.51e5	2.15e6/ 6.06e5
	Memory size	4.40	5.62	8.16	18	39	83
	generation time	00:02	00:05	00:17	01:27	04:18	09:48
	Verification time	00:00,5	00:01	00:04	00:12	00:29	00:53
<i>F-Check</i>	Reachable states	2.71e3	1.07e4	1.71e5	4.38e7	1.12e10	2.87e12
	Memory size	3.26	3.84	4.32	7.60	9.21	15.37
	Verification time	00:07	00:09	00:11	00:21	00:27	0:43

CADP performs very well when communications are not expanded, and in the expanded model, 16 bits seems to be the threshold size where the symbolic model of FormalCheck outperforms the LTS enumeration of CADP.

Table 18.4. Varying Control size

Size of control		2	4	8	16	24	32
a) Measures without expansion of communications							
<i>IF/CADP</i>	Size of LTS trans / states	0.42e2/ 0.2e2	0.62e2/ 0.32e2	1.82e2/ 0.52e2	5.84e2 0.98e2	1.26e3/ 1.46e2	2.25e3/ 1.96e2
	Memory size	8.17	8.18	8.18	8.19	8.22	8.24
	generation time	00:00,5	00:00,6	00:00,8	00:00,9	00:01	00:01
	Verification time	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1
b) Measures with expansion of communications							
<i>IF/CADP</i>	Size of LTS: trans / states	1.86e4/ 5.66e3	6.37e4/ 1.68e4	6.65e5/ 2.01e4	2.01e5/ 6.08e4	0.07e5/ 1.23e5	— —
	Memory size	4.40	5.35	5.87	9.81	19	
	generation time	00:02	00:06	00:08	00:35	01:35	—
	Verification time	00:00,6	00:01	00:01	00:04	00:08	—
<i>F-Check</i>	Reachable states	2.71e3	8.62e3	1.33e6	4.30e10	9.92e23	4.75e30
	Memory size	3.26	4.45	7.54	14.04	24.85	38.89
	Verification time	00:07	00:10	00:10	00:41	10:31	21:36

Table 18.5 gives the results obtained in fixing the data size to a minimum, and in augmenting the number of outputs that are concurrently written (Figure 18.5.b). Clearly, the amount of concurrent paths is hardest on a system that enumerates all possible event inter-leavings. The memory needed for model generation is the limiting performance factor: for instance, in the expanded case, for 8 concurrent output writings, the construction was manually halted after two days, as all the time was spent on swap. In the case of this experiment, the clear performance advantage obtained with FormalCheck should be weighted against the fact that the pseudo-synchronous model benefits from a pre-order reduction of events inter-leavings, optimization that was not performed on the IF model processed by CADP. These experiments are one more demonstration that brute force model checking will not provide satisfactory answers to the problem of asynchronous verification. As could be expected, essential properties should be checked at the highest possible specification level, and the verification engineer should be prepared to apply a variety of reduction techniques to fight the model size explosion.

The next section shows the application of these principles to the design of an asynchronous filter.

*Table 18.5. Varying concurrent paths number*

concurrent paths		1	2	4	8	16	24
a) Measures without expansion of communications							
<i>IF/CADP</i>	Size of LTS: trans / states	0.36e2/ 0.18e2	0.42e2/ 0.2e2	0.98e2/ 0.46e2	2.08e3/ 5.26e2	1.05e6/ 1.31e5	— —
	Memory size	8.17	8.17	8.19	8.77	17	—
	generation time	00:00,5	00:00,5	00:00,9	00:15	01:19	—
	Verification time	00:00,1	00:00,1	00:00,2	00:00,2	00:15	—
b) Measures with expansion of communications							
<i>IF/CADP</i>	Size of LTS: trans / states	3.2e4/ 8.78e3	4.28e4/ 1.13e4	5.94e5/ 1.11e5	? / ?	— —	— —
	Memory size	4.21	5.02	13	>1.2GB	—	—
	generation time	00:03	00:04	00:48	>156:00	—	—
	Verification time	00:01	00:01	00:11	—	—	—
<i>F.CheCk</i>	Reachable states	5.38e3	5.38e3	1.07e4	1.71e5	4.38e7	1.1e10
	Memory size	4.23	3.76	5.56	6.52	11.21	12.94
	Verification time	00:05	00:05	00:07	00:09	00:13	00:22

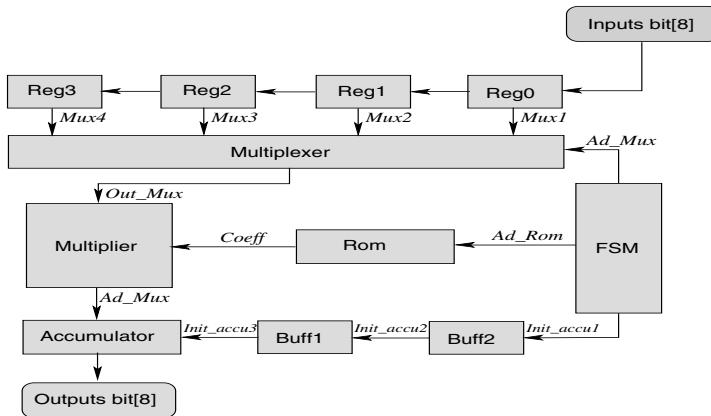


Figure 18.6. Filter behavioral architecture

#### 4. Case study: a four-tap FIR Filter

Figure 18.6 shows a four-tap FIR filter, with 8-bit input/output data, and 4 coefficients. This filter is defined by the following transfer function:

$$Output(n) = \sum_{i=0}^{N-1} Coeff(i)Input(n - i) \quad (N = 4)$$

The filter behavior is modeled by 13 CHP processes, whose structure is shown on Fig. 18.6. FSM and Accumulator contain memorizing elements and are each modeled by two processes for synthesis purposes [14].

##### 4.1 Modeling the Filter in IF

The CHP Component is described by an IF system, and each CHP process is represented by an IF process. To illustrate the translation, we consider one typical process: *Multiplexer* (Figure 18.7). In this process, the control channel (*Ad\_mux*) is typed  $MR[4][1]$ , i.e. one-of-four data encoding. Channel *Ad\_mux* is read in the local variable "address". According to the value of "address", one of the four channels (*Mux1*, *Mux2*, *Mux3* or *Mux4*) is read and its value is written on channel *out\_mux*.

##### 4.2 Some verified properties

Characteristic properties of the filter behavior have been expressed in mu-calculus, and automatically verified using CADP, on a SUN Ultra

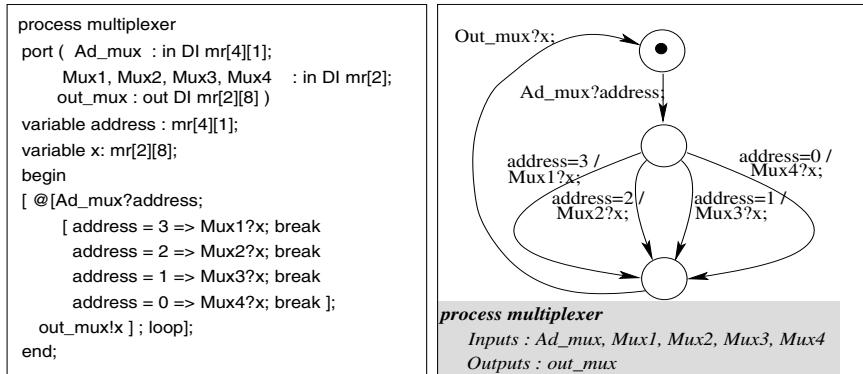


Figure 18.7. the CHP code and the IF representation of Multiplexer.

250 with 1.6 GB memory. For some of them, their meaning, and overall verification time (LTS generation time + model checking time) are listed below.

**P1:** Freedom from deadlock; Verification time:  $1,77 + 7,27 = 9,04$  sec

**P2:** Every input triggers an output; Verification time:  $1,77 + 9,68 = 11,45$  sec

**P3:** The Multiplexer must receive the inputs (Muxi) in the following order: Mux4, Mux3, Mux2 and Mux1; Verification time:  $1,77 + 30,55 = 32,32$  sec

**P4:** No new input is read before Mux1 is read (convolution rule); Verification time:  $1,77 + 8,67 = 10,44$  sec

### 4.3 Verification by behavior reduction

To verify Property P3, which relates to the Mux(i) channels only, an alternative technique is available.

The filter behavior is reduced by hiding all the labels which do not relate to the Mux(i) channels. This can be obtained by applying property-preserving equivalence reduction techniques. The resulting LTS is depicted on Figure 18.8, which shows that the Multiplexer reads the Mux(i) inputs in the correct order.

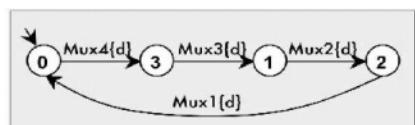


Figure 18.8. Reduced LTS for P3

## 4.4 Handling state explosion

The following techniques have been used during the verification experiments:

*Data abstraction*: this is a well known strategy, by which data are reduced to one bit, when their value do not influence the property at hand. *Explicit generation of interleaving* for CHP intra-process concurrent statements (instead of generating synchronized concurrent IF processes, see section 2.5). One order of magnitude was gained in LTS size (from 2,9 e+5 to 7,4 e+4 states, from 1,5 e+6 to 3,2 e+5 transitions).

## 5. Conclusion

We have implemented a prototype translator that automatically produces the IF model for a CHP specification, along the principles explained in this paper. Preliminary experiments have shown that the IF/CADP toolbox offers a convenient abstraction level for the formal validation of initial CHP specifications. Essential properties can be proven on the specification, before synthesis decisions are made visible in the design description. This provides a new service to the TAST user. During the architecture exploration process, the designer may use transformations that have not been formally proven correct, and wishes to check that essential properties are retained on the refined architecture; our automatic link to IF/CADP is a possible answer to this requirement.

The perspectives of this work include some improvements to the current version of the translator (e.g. negative numbers are currently not recognized) and its application to many more benchmarks. The scalability of the approach to large circuits, of the size of a 32-bit microprocessor will certainly involve elaborate model reduction strategies, some of which are still not automated. Finally, replacing the mu-calculus, the current property notation formalism in CADP, by a more widely accepted property specification language such as the Accelera PSL would ease the designer's access to the verification toolbox.

## References

- [1] M. Renaudin. Asynchronous circuits and systems : a promising design alternative. In *MIGAS 2000, special issue Microelectronics-Engineering Journal*, volume 54, pages 133–149. Elsevier Science, December 2000.
- [2] A.J. Martin. Programming in vlsi: from communicating processes to delay-insensitive circuits. In *Developments in Concurrency and Communication, UT Year of Programming Series*, pages 1–64.

- Addison-Wesley, 1990.
- [3] L. Fesquet Anh Vu Dinh Duc and M. Renaudin. Synthesis of qdi asynchronous circuits from dtl-style petri-net. In *IWLS'02, 11th IEEE ACM Internat. Workshop on Logic and Synthesis, New Orleans*, volume 55. Elsevier Science, June 2002.
  - [4] M. Renaudin, J. Rigaud, A. Dinhduc, A. Rezzag, A. Siriani, and J. Fragoso. Tast cad tools. In *ASYNC'02 TUTORIAL*, 2002. ISRN: TIMA-RR-02/04/01-FR.
  - [5] M. Renaudin, P. Vivet, and F. Robin. Aspro-216 : a standard-cell q.d.i. 16-bit risc asynchronous microprocessor. In *Proc. of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'98), San Diego - USA*, pages 22–31, 1998.
  - [6] P. Senn A. Abrial, M. Renaudin and P. Vivet. A new contactless smart card ic using on-chip antenna and asynchronounous microcontroller. *Journal of Solid-State Circuits*, 36:1101–1107, 2001.
  - [7] D. Borrione, M. Boubekeur, E. Dumitrescu, M. Renaudin, and J. Rigaud. An approach to the introduction of formal validation in an asynchronous circuit design flow. In *Proc. 36th Hawai Int. Conf. on System Sciences (HICSS'03)*, pages 1101–1107, Jan 2003.
  - [8] S. Graf M. Bozga and L. Mounier. Automated validation of distributed software using the if environment. In *Proc. Workshop on Software Model-checking*, volume 55. Elsevier Science, July 2000.
  - [9] M. Bozga, J. Fernandez, S. Graf, and L. Mounier. If : An intermediate representation and validation environment for timed asynchronous systems. In *Proc. FM'99, Toulouse*, 1999.
  - [10] <http://www.inrialpes.fr/vasy/cadp/>.
  - [11] K. Van Berkel. Handshake circuits - an asynchronous architecture for vlsi programming. volume 55. Cambridge University Press, July 1993.

## Chapter 19

# COMBINED FORMAL REFINEMENT AND MODEL CHECKING FOR REAL-TIME SYSTEMS VERIFICATION\*

Alexander Krupp<sup>1</sup>, Wolfgang Mueller<sup>1</sup>, Ian Oliver<sup>2</sup>

<sup>1</sup>*Paderborn University, Paderborn, Germany*

<sup>2</sup>*Nokia Research Centre, Helsinki, Finland*

**Abstract** We present a framework, which combines model checking and theorem prover based refinement for real-time systems design focusing on the refinement of non deterministic to timed deterministic finite state machines. Our verification flow starts from a cycle accurate finite state machine for the RAVEN model checker. We present a translation, which transforms the model into efficient B language code. After refining the RAVEN model and annotating it, the time accurate model is also translated to B so that the B theorem prover can verify the refined model almost automatically. The approach is introduced by the example of a mobile phone echo cancellation unit.

## 1. Introduction

With the increasing complexity of systems on a chip, formal verification is of increasing importance. In several fields of hardware design, equivalence checks and model checking are applied on a regular basis. For software design, the B method has been successfully applied in several industrial projects. Additionally, model checking receives increasing acceptance in embedded software design. However, theorem proving in general is less accepted since it still requires too many user interactions conducted by educated experts.

\*The work described herein is funded by the IST Project PUSSEE

This article describes an efficient combination of model checking and theorem proven refinement based on the RAVEN model checker [13] and the Atelier-B B toolset [1]. We present a model checking oriented verification flow, which is based on automatic translation to B for refinement verification. We focus our interest on the refinement of a cycle accurate model into a time accurate model, which might be further refined to an implementation. In that context, we present a B code generation for a very efficient application of the Atelier-B theorem prover. In contrast to related approaches, our experimental results demonstrate that the proof of the generated code requires almost no interaction with the prover and additionally gives very low runtimes.

The remainder of this article is structured as follows. The next section discusses related works. Section 3 and 4 introduce RAVEN and B before Section 5 outlines our approach by the examples of the refinement of an echo cancellation unit. Thereafter, we present our experimental results before the final section closes with a conclusion.

## 2. Related Work

There is significant work integrating model checkers into theorem provers or vice versa. PVS (Prototype Verification System) is a theorem prover where the PVS *specification language* is based on high order predicate logic. Shankar et al. enhance PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking to separate concerns as well as to cover concurrent systems properties [14]. STeP (Stanford Temporal Prover) is implemented in Standard ML and C [9] and integrates a model checker into an automatic deductive theorem prover. The input for model checking is given as a set of temporal formulae and a transition system, which is generated from a description in a reactive system specification language (SPL) or a description of a VHDL subset. The SyMP framework integrates a model checker into a HOL based theorem prover for general investigations on effectiveness and efficiency [4]. The work focus on computer assisted manual proofs where main examples come from hardware design. Mocha [2] is a model checker enhanced by a theorem prover and a simulator to provide an interactive environment for concurrent system specification and verification. However, in the Mocha framework theorem proving is interactive and no efficient reasoning of more complex systems is reported.

In the context of the B theorem prover, Mikhailov and Butler combine theorem proving and constraint solving [10]. They focus on the B theorem prover and the Alloy Constraint Analyser for general property

verification. Fokkink et al. employ the B method and combine it with  $\mu$ CRL [3]. They describe the use of B refinement in combination with model checking to arrive at a formally verified prototype implementation of a data acquisition system of the Lynx military helicopters. They present the refinement of a system implementation starting from a first abstract property specification.

All those approaches consider timeless models and do not cover refinement with respect to real-time properties in finite state machines. Only Zandin has investigated real-time property specification with B by the example of a cruise controller[15]. However, he reports significant problems with respect to the complexity of the proof during refinement.

In contrast to the HOL based approaches, we present a model checking based approach with the RAVEN model checker in conjunction with the Atelier-B theorem prover for formal refinement with very little user interaction. We focus on the verification of real-time systems and on the refinement from cycle accurate models to time accurate models based on an efficient mapping from the RAVEN Input Language to the B language.

### 3. Real-Time Model Checking with RAVEN

BDD based model checking in the domain of electronic design automation is due to the pioneering work of Clarke et al. in [6] and the SMV model checker. SMV verifies a given set of synchronously communicating state machines with respect to properties given by a set of formulae in tree temporal logic, namely CTL (Computational Tree Logic).

For our approach, we apply the RAVEN (Real-Time Analyzing and Verification Environment) real-time model checker, which extends basic model checking for real-time systems verification by additional analysis algorithms [13]. In RAVEN, a model is given by I/O-Interval Structures and the specification by CCTL (Clocked CTL).

I/O-Interval Structures are based on Kripke structures with  $[\min, \max]$ -time intervals at the state transitions. Executing interval structures proceeds as follows. Assume that each interval structure has exactly one clock for measuring time. The clock is reset to zero, if a new state is entered. A state can be left, if the actual clock value corresponds to a delay time labelled at an outgoing transition. The state must be left when the maximal delay time of all outgoing transitions is reached.

CCTL is a time bounded temporal logic. In contrast to classical CTL, the temporal operators **F** (i.e., eventually), **G** (globally), and **U** (until) are provided with interval time bounds  $[a, b]$ ,  $a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\}$ . The symbol  $\infty$  is defined through:  $\forall i \in \mathbb{N}_0 : i < \infty$ . Those temporal

operators can have no or just a single time bound. If no value is defined, the lower bound is zero and the upper bound is infinity by default. The **X**-operator (i.e., next) can have a single time bound  $[a]$  only ( $a \in \mathbb{N}$ ). If no time bound is specified, it is implicitly set to one.

In RAVEN, I/O-Interval Structures and a set of CCTL formulae are specified by means of the textual RAVEN Input Language (RIL). A RIL specification contains (a) a set of global definitions, e.g., fixed time bounds or frequently used formulae, (b) the specification of parallel running modules, i.e., a textual specification of I/O-Interval Structures, and (c) a set of CCTL formulae, representing required properties of the model. The following example is a RIL module with 4 states  $s \in \{\text{wait}, \dots\}$  and two state transitions from *wait* to *failed* and to *accept*.

```

MODULE consumer
SIGNAL s: {wait,reject,accept,failed}
INPUTS loadFail := GlobalFailure
      loaderIdle := (loader.state=loader.idle)
DEFINE rejectOrder := (s = reject)
INIT   s = wait
TRANS  |- s=wait
      -- loadFail          --> s:=failed
      -- !loadFail & loaderIdle --> s:=accept
...

```

For property specification, consider the following CCTL formula. It defines that the input buffer of a consumer must not be blocked to guarantee a sufficient workload, i.e., each accepted delivery request must be followed by loading an item at the input within 100 time units after acceptance:

```

AG((consumer.s = consumer.accept)
 -> AF[100]((loader.state = loader.wait)& AX(loader.state = loader.load)))

```

## 4. Refinement with B

The two classical approaches to theorem proving in the domain of electronic design automation are the Boyer–Moore Theorem Prover (BMTP) and HOL. BMTP and HOL are both interactive proof assistants for higher order logic [5, 8]. In theorem proving, a proof has to be interactively derived from a set of axioms and inference rules. Though several practical studies have been undertaken, classical interactive theorem proving has not received a wide industrial acceptance so far.

In the domain of software oriented system, B was introduced by Abrial in [1] as a methodology, a language, and a toolset. It has been successfully applied in several industrial projects, e.g., in the Parisian driverless metro Meteor in 1992.

Similar to Z, B is based on viewing a program as a mathematical object and the concepts of pre-and postconditions, of non determinism, and weakest precondition. The B language is based on the concept of abstract machines. An abstract machine consists of VARIABLES and OPERATIONS and is defined as follows

```
MACHINE M( ... )
  CONSTRAINTS
  ...
  VARIABLES
  ...
  INVARIANT
  ...
  INITIALISATION
  ...
  OPERATIONS
  ...
```

Variables represent the state of the B machine, which is constrained by the INVARIANT. Operations may change the machine's state and return a list of results. A B abstract machine is refined by reducing non determinism and abstract functions until a deterministic implementation is reached, which is denoted as  $B_0$ .  $B_0$  is a B subset and can be translated into executable code of a programming language.

Refinement in B means the replacement of a machine  $M$  by a machine  $N$  where the operations of  $M$  are (re)defined by  $N$ . Syntactically, a refinement is given by

```
REFINEMENT N REFINES M
  ...
```

In  $N$ ,  $M$  operations have to be given by an identical signatures. However, they may refer to different internal states or even to a different definition. The important requirement is that the user must be able to use machine  $N$  as machine  $M$ .

Refinement can be conducted by (i) the removal of preconditions and non deterministic choices, (ii) the introduction of control structures, e.g., sequencing and loops, and (iii) the transformation of data structures. A final refinement in  $B_0$  defines the implementation of the system as

```
IMPLEMENTATION I REFINES M
  ...
```

At implementation level, the B definition looks very much like a PASCAL program from which a B toolset can automatically generate Ada, C, or C++.

## 5. Combined Model Checking and Refinement

Systems' design today typically starts with state oriented models. For several applications, StateCharts are frequently applied as an advanced

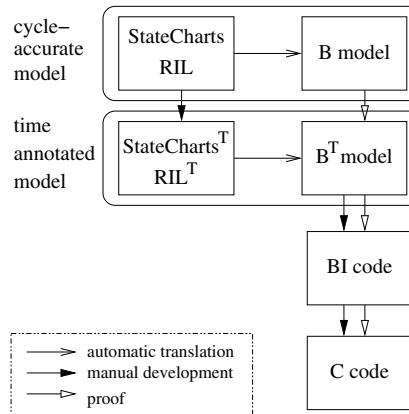


Figure 19.1. Verification Based Design for Real-Time Systems Refinement

graphical front-end for documentation, model checking, as well as for VHDL, Verilog, and C code generation.

We present a design flow for proven design based on the combined application of model checking and formal refinement. In our verification environment, we apply StateCharts as a graphical capture and generate RAVEN RIL code from it. At that level, we can investigate the cycle accurate models by simulation and checking their properties through model checking by the RAVEN model checker. After first verifications, we automatically generate a corresponding B code model to prove the correctness of the further refinements. Here, our focus is on the refinement of the RIL model through annotation of transitions with timing specification and minor modification in state transitions like removal of self loops. The time annotated RIL model is denoted as  $RIL^T$ . When generating B code from  $RIL^T$  resulting in  $B^T$ , the B prover is applied to verify if the  $B^T$  model is a refinement of the B model, which also verifies that the  $RIL^T$  model is a correct refinement of the RIL model. In the ideal case that step should be automatically. However, practice shows that this needs a lot of user interaction and any help for refinement automation is appreciated. Through the B environment, further manual refinement to the B implementation level B0 is possible, from which we can automatically generate C code for a correct implementation.

Figure 19.1 gives an overview of the complete design flow from StateCharts over RIL to time annotated  $RIL^T$  and, correspondingly, over B and  $B^T$  to C code generation. To illustrate the approach, we employ an example from the industrial case study of an echo cancellation unit of a mobile phone.

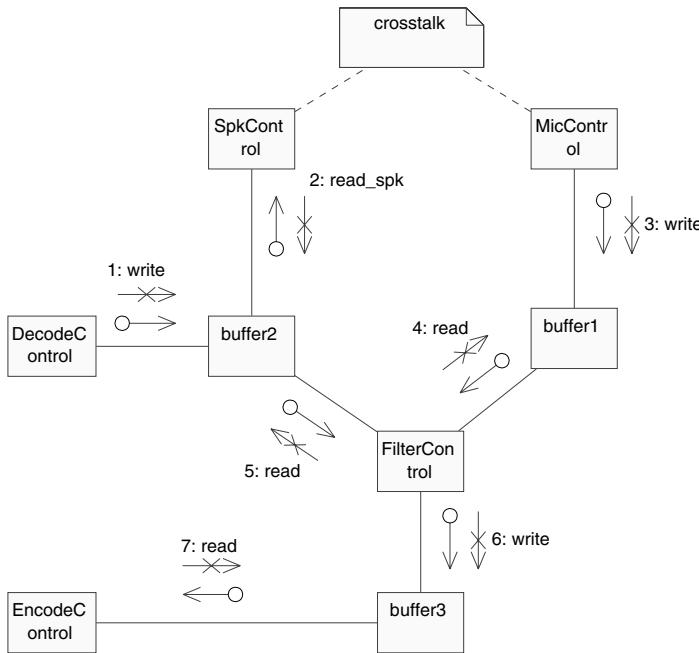


Figure 19.2. Collaboration diagram of echo cancellation

## 5.1 The Echo Cancellation Unit

Mobile phones contain an echo cancellation unit (ECU) to filter cross-talk stemming from the phone's speaker interfering with the audio picked up at the microphone - this is basically to prevent the feedback loop between the speaker and the microphone while the user is engaged in conversation. To prevent this feedback loop, crosstalk has to be suppressed in the audio packets generated from the microphone. A digital filter, which correlates received input data with the speaker's output audio data, performs the suppression. For echo cancellation of recorded audio, the audio I/O interface decodes data from the network and generates output to speaker. Simultaneously, data from the microphone are recorded, filtered, and generated for output to the network.

The model of the ECU can be reduced to two producers, two consumers, three buffers, and a filter, which can be considered as a producer and a consumer again. Data and message flow is shown in the UML collaboration diagram in Figure 19.2. A decoding process *DecodeControl* decodes audio data packets received from the network and inserts a packet into a buffer *buffer2*. From that buffer, audio packets

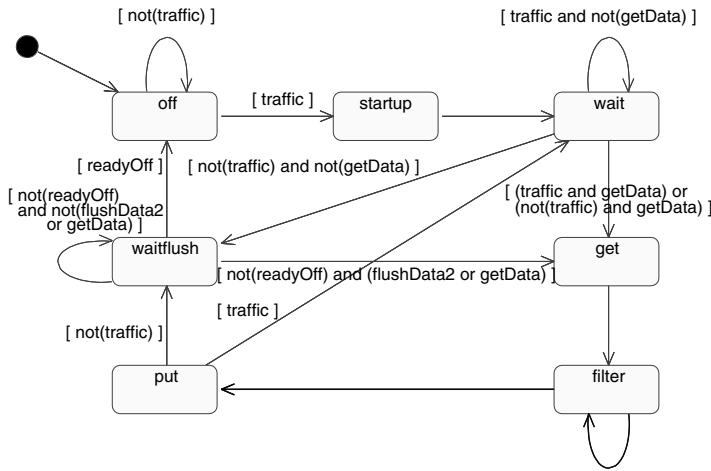


Figure 19.3. StateChart FilterControl\_BEHAVIOUR

are given to a speaker through *SpkControl*. At the same time, *MicControl* encodes audio from a microphone to packets, which are inserted into another buffer, *buffer1*. The filter process is called *FilterControl* here. It receives a packet from the speaker buffer *buffer2* after it has been given to the speaker and uses it to filter a packet taken from the microphone buffer *buffer1*. The filtered packet is passed to *buffer3*. From there it is consumed by the network encoding *EncodeControl* and passed to the network. In that process, buffer overflows as well as buffer underflows are harmful. In the remainder of this sections we focus on the refinement of the *FilterControl* component to outline our approach.

## 5.2 RIL Code Generation

Our verification flow starts with a UML StateChart model, which is translated into a RIL presentation for model checking. We apply a Statemate style semantics to a subset of UML StateCharts and process the RIL code generation corresponding to the SMV code generation presented in [7]. The state hierarchies are mainly mapped into RIL modules and their connections between modules through INPUTs, and DEFINEs. For state transitions, guards are limited to Boolean and Integer variables where actions define their updates. That is basically due to limitations in model checking and formal refinement, which both do not efficiently apply to more complex data types.

The *FilterControl\_BEHAVIOUR* StateChart of the *FilterControl* component (Figure 19.3) models both a producer and consumer and performs

as follows. A transition from `off` to `startup` is executed when input traffic is detected. Thereafter, state `wait` is entered. A self loop in the diagram indicates, that state `wait` is kept while no data is available in traffic mode. When traffic mode is turned off and no data available, `waitflush` is entered. In the case of available data, state `get` is always entered. From `get` a transition leads to state `filter` from which a non-deterministic transition to `put` and `filter` is given. The non-determinism is introduced here because we want to leave the exact timing open. If input traffic is enabled in state `put`, a transition to state `wait` is performed. Finally, `waitflush` flushes the buffer. From there, the system can change to `off` or to `get` as soon as the two buffers are empty and the microphone, the decoding module, and `FilterControl` are turned off. If data are available again, the system changes goes to `get` and starts the next cycle. Please note, that we apply *guards* instead of events on transitions.

### 5.3 B Generation

Transforming RIL models into B machines requires basic restructuring of the modules. For an efficient proof, we break down the generated code into two B specifications, i.e., two for each RIL module. The first one covers structural definitions and the second one adds behavioural definitions. The separation into two B machines makes these machines better manageable by the B prover, which generates much less proof obligations for each machine. The structural B machines cover the data dependencies as well as the declaration of state variables and their range. Invariants additionally define data consistencies, which are propagated between the different B machines.

The B machines covering the module's behaviour define the corresponding guarded state transition and their actions, which are mapped to one operation *doTransition*. The following code fragment gives a simple example of a state machine as shown in Figure 19.3. The transition from *s=off* to *s=startup* is guarded by a Boolean variable *traffic*; an unguarded transition goes from *s=get* to *s=filter*, where transitions from *s=startup* are just sketched. From *s=filter* we define a non-deterministic choice between a self-loop transition and a transition to *s=put*.

```

REFINEMENT FilterControl_BEHAVIOUR
REFINES FilterControl_STRUCTURE
...
OPERATIONS
...
doTransition =
BEGIN
  IF (s = off)      THEN
    SELECT ( traffic = TRUE ) THEN s := startup

```

```

        ELSE s := off
    END
ELSIF (s = startup) THEN ...
ELSIF (s = get)      THEN SIGNAL_s := filter ...
ELSIF (s = filter)  THEN
    SELECT (TRUE = TRUE) THEN s := put
    WHEN   (TRUE = TRUE) THEN s := filter
END
ELSIF (SIGNAL_s = put ) THEN
...
END
END

```

The *doTransition* operations are triggered by one extra B machine, i.e., EXEC, which implements the computation model of RAVEN's I/O-Interval Structures. The EXEC machine basically implements two computational steps:

1. value propagation (RIL DEFINEs) to corresponding INPUTs of other modules
2. transition execution through *doTransition*

Additional invariants in the EXEC machine guarantee that those steps are executed in a specific order. Due to technical matters we require yet another B machine, which starts computation but which is ignored in the remainder of this article.

## 5.4 RIL Refinement

When generating RIL from StateCharts, we arrive at synchronously communicating non deterministic finite state machines, which are executed at cycle accurate basis.

For real-time system specification and verification, those state machines are annotated in RIL with timing information and transitions are optimised and specified in details so that we obtain time annotated, deterministic finite state machines and denote them as RIL<sup>T</sup>. Consider the following example of a non deterministic transition from state *s=filter* to *put/filter*.

```
|- s=filter --  --> s := put
                  --> s := filter
```

The transition can be refined to a timed transition with time 2 such as

```
|- s=filter -- :2 --> s := put
```

This transition fires after 2 time steps and changes to state *s=put*. It remains in *filter* until the timer expires. Because the transition does

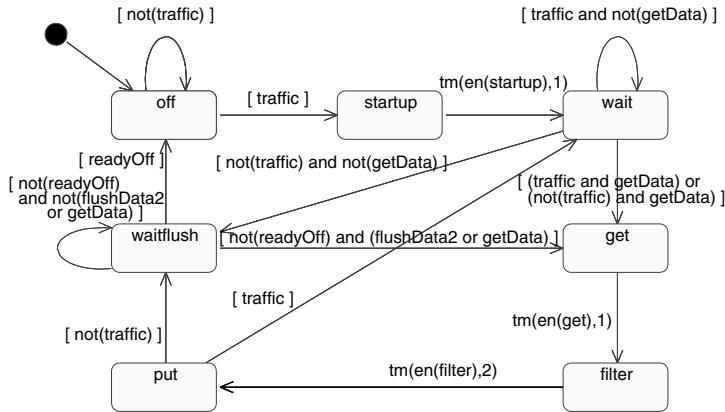


Figure 19.4. StateChart *FilterControl\_TBEHAVIOUR*

not lead to a state other than *put* or *filter*, it implements an actual refinement of the previous specification. The timed StateChart *FilterControl\_TBEHAVIOUR* (figure 19.4) introduces this timed transition graphically. The notation is derived from STATEMATE, where  $tm(en(a), 9)$  means, that the event fires 9 time steps after state *a* is entered.

## 5.5 $B^T$ Generation

Starting from RIL $T$ , we can automatically generate the corresponding B code, i.e.,  $B^T$ , as a refinement of the previously generated B so that the B prover can verify the refinement step. The  $B^T$  code generation performs similar to the B code generation with additional instantiation of a Timer and TIME constants for each machine and the additional code in *doTransition* for managing (advancing and resetting) the timer.

Consider the ECU example and the refinement of the previous subsection by adding a time delay of 2 to the transition. The following code defines *FilterControl\_TBEHAVIOUR* as a refinement of *FilterControl\_BEHAVIOUR*. The refinement instantiates a timer with delay 2. In the transition from *s=filter*, the transition definition is encapsulated and executed only when the timer has expired. Otherwise, the execution is skipped. If expiring, the timer is reset, otherwise it is advanced until it reaches its upper bound.

```

REFINEMENT FilterControl_TBEHAVIOUR
REFINES FilterControl_BEHAVIOUR
...
OPERATIONS
...
doTransition =

```

```

BEGIN
  IF ( s = off ) THEN
  ...
  ELSIF ( s = filter ) THEN DELAY 2 THEN s := put END
  ELSIF ( s = put ) THEN ... END
END

```

With the given B behavioural model and its time annotated refinement, the B system now can generate proofs, which are verified by the B prover.

## 5.6 $B^T$ Refinement and C Code Generation

Refining  $B^T$  to  $B_0$  basically means replacing simultaneous by sequential substitution and replacing non deterministic predicates and statements by deterministic ones. This results in substantial changes of the example operation code. From a programmer's point of view the operation *doTransition* description is now very straight forward, as the semantics of the  $B_0$  language is very similar to traditional sequential programming languages where the state transitions are given by embedded if-then-else statements.

```

IMPLEMENTATION FilterControl_B0
REFINES FilterControl_TBEHAVIOUR
IMPORTS TIMER_T1.Timer(2)

...
doTransition =
BEGIN
  IF ( s = off ) THEN
  ...
  ELSIF ( s = filter ) THEN
    IF TIMER_T1.expired = TRUE THEN s := ready; TIMER_T1.doReset
    ELSE s := s; TIMER_T1.doAdvance
  END
  ELSIF s = put THEN ...
  ...
END

```

At that level, the code is well structured and can be easily managed by a person with programming skills. The proof of the implementation for  $B_0$  can be automatically accomplished with *Atelier-B*. After successful verification, proven C code can be automatically generated.

## 6. Experimental Results

We have tested several variations for generating B code from RIL and their refinement with *Atelier-B* before arriving at the approach presented in the previous section. First trials gave huge numbers of generated proof obligations (proofs automatically generated by Atelier-B) with up to 90.000 proof obligations, resulting in a not acceptable runtime of more

than 22 hours on a Sun Enterprise 450/4400. Those experiments have shown that successful proof of the composed system in B very much depends on the separation of the specification into different B machines so that we finally arrived at the presented efficient B patterns and the separation into the presented levels.

We have exercised the approach with the presented case study at Nokia Research Centre. The verification was performed under Linux on an Intel 2.2GHz P4. The time for B proof obligation generation was 271 sec. The proof of the 62 proof obligations in automatic *Force 1* mode took 20 sec. Almost the entire proof was accomplished by *Atelier B* automatically. Just one single proof obligation (PO) could not be proven by the automatic prover. It was the B machine, which was introduced as an extra component to trigger the timeouts. The proof of that PO was performed through one simple command of the interactive prover of Atelier-B, which took less than one second for manual invocation. Thereafter, the refinement of the echo cancellation unit example was completely proven by *Atelier B*.

The following table gives refinement details of the example with the number of obvious and non obvious proof obligations (POs), which were generated and proven at each level.

Level	No. of Modules	Obvious POs	Non Obvious POs
B (Structure)	12	388	10
B (Behaviour)	5	605	0
$B^T$	7	1843	52
Total	24	2836	62

## 7. Conclusions

A novel approach to systems design has been investigated, defining the integrated tool oriented application of formal verification through model checking and formal refinement by deduction. We have successfully applied this approach to ‘Model Based Testing’ in the design of certain embedded systems [11, 12]. Two tools have been applied in their corresponding domain of formal verification. The RAVEN model checker from the University of Tübingen with its input language RIL has been applied in combination with a tool for deductive refinement, namely the B prover Atelier-B from Clearsy. Our experiments have demonstrated that model checking in combination with refinement by deduction can be successfully applied for real-time systems design, i.e., for refinement from a non deterministic, cycle accurate to a deterministic time annotated model. After testing various alternatives, we found an efficient translation from RIL to B where the generated B code can be efficiently

proven by Atelier-B without any major manual interference. However, though the results are promising, we are aware that it is too early to draw general conclusions for other domains since it has also been shown that little variations in the B model easily have big effects on the prover and the number of generated proof obligations.

## References

- [1] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.
- [2] R. Alur and T.A. Henzinger. Reactive Modules. In *LICS'96*, 1996.
- [3] S. Blom et al.  $\mu$ CRL: A Toolset for Analysing Algebraic Specifications. In *Proc. of CAV'01*, 2001.
- [4] S. Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.
- [5] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.
- [6] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. *Lecture Notes in Computer Science*, 131, 1981.
- [7] E.M. Clarke and W. Heinle. Modular Translation of Statecharts to SMV. Technical Report, Carnegie Mellon University, 2000.
- [8] M.J. Gordon. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
- [9] Z. Manna et al. STeP: The Stanford Temporal Prover. Technical Report, Stanford University, 1994.
- [10] L. Mikhailov and M. Butler. An Approach to Combining B and Alloy. In D. Bert et al., editors, *ZB'2002*, Grenoble, France, 2002.
- [11] I. Oliver. Experiences of Model Driven Architecture in Real-Time Embedded systems. In *Proc. of FDL02, Marseille, France*, 2002.
- [12] I. Oliver. Model Driven Embedded Systems. In J. Lilius et al., editors, *Proc. of ACSD2003, Guimaraes, Portugal*, 2002.
- [13] J. Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *J.UCS, Springer, Heidelberg*, 2001.
- [14] N. Shankar. Combining Theorem Proving and Model Checking Through Symbolic Analysis. In *CONCUR 2000*, 2000.
- [15] J. Zandin. Non-Operational, Temporal Specification Using the B Method - A Cruise Controller Case Study. Master's Thesis, Chalmers University of Technology and Gothenburg University, 1999.

# Chapter 20

## REFINEMENT OF HYBRID SYSTEMS \*

*From Formal Models to Design Languages*

Jan Romberg<sup>1</sup>, Christoph Grimm<sup>2</sup>

<sup>1</sup>*Systems & Software Engineering, TU München, Germany*

<sup>2</sup>*Technische Informatik, J. W. Goethe-Universität Frankfurt am Main, Germany*

**Abstract** System-level design of discrete-continuous embedded systems is a complex and error-prone task. While existing design languages like SystemC and its extension to the mixed-signal domain, SystemC-AMS, are supported by tools and libraries, they lack both mathematical precision and intuitive, abstract design notations. Graphical design notations with formal foundations such as HyCharts suffer from the lack of tool support and acceptance in the developer community. To overcome the deficiencies of both approaches, we present a design flow from graphical HyCharts to SystemC-AMS designs. This design flow uses a formally founded refinement technique to ensure the overall consistency of the design.

### 1. Introduction

Embedded control systems are frequently characterized as a mixture of continuous and discrete behaviors. We call such systems discrete-continuous or *hybrid* systems. Because the discrete part of a hybrid system introduces discontinuities in the system's evolution, the behavior of hybrid systems tends to be difficult to predict and analyze. For the task of high-level design of a control system, it is highly desirable to use representations that accurately reflect both continuous and discrete

\*This work was supported with funds of the Deutsche Forschungsgemeinschaft under reference numbers Br 887/9 and Wa 357/14-2 within the priority program *Design and design methodology of embedded systems*.

behavior. In the early stages of a design, this frees the developer from considering implementation details like quantization and sampling, and allows designers to concentrate on the essential features of the design.

Formalisms like Hybrid Automata [1], or HyCharts [11] are well suited for precisely capturing the continuous/discrete behavior of a hybrid system. An advantage of these formalisms is that, being based on formal semantics, models are susceptible to automated analysis and formal verification. However, as most designs are implemented using digital hardware, there is currently a gap between the capturing and verification of an abstract design in mixed discrete-continuous time, and the discretized design and implementation of such systems. For the later design phases, discrete *approximations* of the hybrid model that explicitly consider quantization and sampling effects are more appropriate.

Hybrid systems combine continuous behavior specified by differential equations with discontinuities introduced by discrete switching logic. With discrete hardware being pervasive in embedded systems, the prevalent way of simulating and implementing hybrid systems is based on discrete-time or discrete-event algorithms. Numerical solvers are one example for such discrete algorithms; a variety of variable- and fixed-step algorithms are successfully used for continuous-time simulation, and their capabilities and limitations are well-understood [9]. For implementation or large simulations, simple fixed-step algorithms like Euler forward with quasi-constant performance are widespread.

The effects that introduce deviations to the ‘ideal’ behavior of a realization (or simulation) can be roughly characterized as:

- Quantization and limitation of the variable’s values.
- Quantization of the time. Modeling smooth changes of analog functions would require an infinite number of events/process activations. This is approximated by activation at a finite number of discrete time steps.

It has been recognized by numerous authors [9] that simulation or real-time computation of hybrid system across discontinuities may cause large errors when the design is discretized ad-hoc. For component-level simulation, variable-step algorithms offer good results; however, simulation performance is generally unacceptable for larger (system-level) designs.

SystemC-AMS [5] offers support for mixed solvers for continuous, hybrid, and discrete designs. Cyclic dependencies between different solver kernels are broken into acyclic structures by introducing a sufficiently small delay. The resulting acyclic structure is then ordered in the direction of the signal’s flow. After that, the outputs can be computed successively from already known inputs by executing a step in each solver

kernel separately. This model of computation is called *static dataflow*, and the delay determines the frequency with which the signals between the signal processing blocks are sampled.

We'd like to employ the approach in the sense that formal system specifications in HyCharts are translated to fixed-step discrete parts in the SystemC-AMS model, and SystemC-AMS's support for mixed-mode simulation is employed for simulating the system along with, for instance, a model of the system's environment.

In this paper, we define a design flow from a formal model of a hybrid system to its discrete realization. We use HyCharts for modeling the hybrid system, and SystemC(-AMS) with static dataflow model of computation for simulation, and as a starting point for hardware/software co-design. Figure 20.1 gives an overview of the proposed design flow. Starting from a HyChart model, a discretized HyChart is derived and shown to be a time refinement of a relaxed version of the original HyChart. The discrete HyChart can then be mapped to an equivalent discrete-time SystemC model. This model, in turn, is used for simulation and synthesis of both analog and discrete HW/SW components.

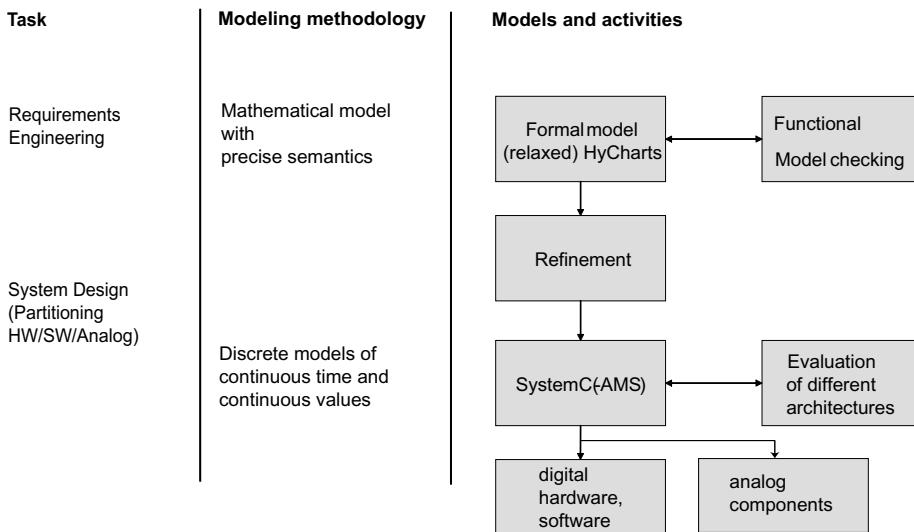


Figure 20.1. Design flow and translation between HyCharts and SystemC

**Related work.** Refinement methodologies are known in different domains: In an informal context [5, 4] an executable specification is augmented with further design details in a series of successive design steps.

After each step, the design is evaluated by simulation. We call this *design refinement*. E. g. in [5], a SystemC model of a block diagram is refined from a continuous time block diagram to a mixed-signal architecture. The effect of each discretization can be validated by a simulation.

Formally, *behavior refinement* may be defined as the containment of the input/output relation of one (concrete) component in the corresponding relation of another (more abstract) component. [3] describes refinement involving a change of the underlying time model. Much of the work in this paper is based on [11] which discusses time refinement, relaxation, and discretization in the context of HyCharts. A related approach for the hybrid formalism *Charon* is described in [2].

This paper is structured as follows: Section 20.2 gives an overview of the HyCharts formalism used for specification of hybrid systems, and presents a method for relaxation and time refinement within this framework. Section 20.3 briefly describes the design and simulation of signal processing systems using SystemC. Section 20.4 explains the translation from a discrete HyCharts model to SystemC. In our view, the methodology is not necessarily restricted to the combination HyCharts/SystemC; a similar method may be derived for other formalisms and design frameworks.

## 2. HyCharts

HyCharts [6] were developed as a graphical notation for hybrid systems, similar in some respect to graphical design notations like UML-RT/ROOM [10], yet formally precise like hybrid automata [1] or related formalisms.

**Hierarchical graphs.** As a common syntactical representation both HyACharts and HySCharts are based on *hierarchical graphs*. Each hierarchical graph is constructed with the following operators:  $\star$  (independent composition of nodes),  $,$  (sequential composition of nodes),  $\uparrow$  (Feedback),  $\bullet <$  (identification),  $\rhd \bullet$  (ramification), and  $\times$  (transposition). The semantics of both structural and behavioral description is defined by associating a meaning with the graph operators.

**Structural description with HyACharts.** A HyAChart (Hybrid Architecture Chart) consists of a hierarchical graph whose nodes represent *components*, and whose arcs represent *channels*. The semantics of the hierarchical graph is *multiplicative*: all the nodes in the graph are concurrently active. Being a hierarchical graph, each node, in turn, may have sub-nodes.

Figure 20.2 shows an example of a HyAChart of an electronic height control system (EHC). The purpose of the EHC, which was taken from an automotive case study, is to control the chassis level of an automobile by means of a pneumatic suspension. A chassis level  $sH$  is measured by sensors and filtered to eliminate noise. The filtered value  $fH$  is read periodically by CONTROL. CONTROL outputs a target chassis level  $aH$  encoding the rate of change due to the operation of two actuators, a compressor and an escape valve.

The basic operations of CONTROL are: (1) if the chassis level is below a certain lower bound, a compressor is used to increase it, (2) if the level is too high, air is blown off by opening an escape valve, (3) whenever the car is going through a curve, the EHC is turned off, (4) whenever the chassis level is back in the tolerance range, the filter is reset.

For the remainder of this paper we will concentrate on the CONTROL component as it contains the essential functionality. Extension of our approach to the other system parts is straightforward.

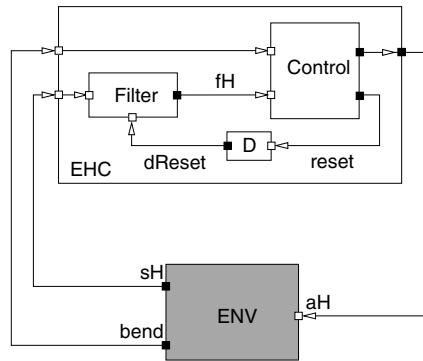


Figure 20.2. HyAChart of the EHC system

**Behavioral description with HySCharts.** HySCharts are also based on hierarchical graphs; the graph used in the semantics is obtained from the specification by a simple syntactic transformation. Graphs for HySCharts are interpreted *additively*: Only one node in the graph is active at a given time.

A HySChart defines the (discrete or continuous) behavior of a component. Nodes (rounded boxes) represent control states, and arcs represent transitions. Similar to the STATECHARTS formalism, HySCharts are hierarchical in the sense that control states may be refined by sub-states. As an example, CONTROL's behavior is specified by the HySChart shown in figure 20.3.

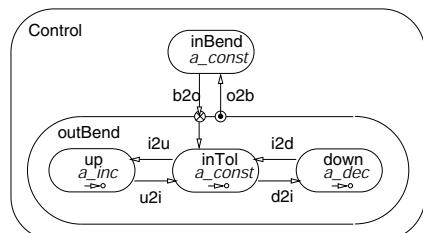


Figure 20.3. HySChart of Control component

CONTROL's HySChart has two levels of hierarchy: The topmost state CONTROL has the sub-states INBEND (car is driving through a curve), and OUTBEND (otherwise). OUTBEND in turn is decomposed into sub-states INTOL (chassis level within tolerance), UP (raise chassis level), and DOWN (lower chassis level).

Transitions in HySCharts are labeled with *actions*. Actions are a conjunction of a precondition on the component's latched state and current input (*guard*) and a postcondition determining the next data state. We use left-quoted variables  $v^*$  for current inputs, right-quoted variables  $v'$  for the next data state, and plain variables for the latched data state. For variables with discontinuous changes, discontinuities are detected using *timestamps*. We write  $v.val^*$  for the current value,  $v.t^*$  for the current timestamp, and  $v.t$  for the latched timestamp of  $v$ . The special variable *now* refers to a global clock evolving continuously with time.

Table 20.1. Actions, invariants and activities for Control

	HySChart	DiSChart
Actions:		
<i>b2o</i>	<i>bend?</i>	
<i>o2b</i>	<i>b2o</i>	
<i>i2u</i>	$fH.val^* \leq lb$	
<i>i2d</i>	$fH.val^* \geq ub$	
<i>u2i</i>	$fH.val^* \geq lb + c \wedge reset!$	
<i>d2i</i>	$fH.val^* \leq ub - c \wedge reset!$	
Invariants:		
$inBend_{inv}$	$bend.t = bend.t^*$ $\vee now - bend.t^* < \epsilon_{bend}$	$bend.t = bend.t^*$ $\vee now \bmod T \neq 0$
$inTol_{inv}$	$fH.val^* \in (lb - \epsilon_{i1}, ub + \epsilon_{i2})$ $\vee now - fH.t^* < \epsilon_{fH}$	$fH.val^* \in (lb, ub)$ $\vee now \bmod T \neq 0$
$up_{inv}$	$fH.val^* < lb + c + \epsilon_u$ $\vee now - fH.t^* < \epsilon_{fH}$	$fH.val^* < lb + c$ $\vee now \bmod T \neq 0$
$down_{inv}$	$fH.val^* > ub - c - \epsilon_d$ $\vee now - fH.t^* < \epsilon_{fH}$	$fH.val^* > ub - c$ $\vee now \bmod T \neq 0$
Activities:		
<i>a_const</i>	$aH = 0$ with $\epsilon_{dis}.aH = 0$	$aH' = aH$
<i>a_inc</i>	$aH = cp$ with $\epsilon_{dis}.aH = 0$	$aH' = aH + cp \cdot T$
<i>a_dec</i>	$aH = ev$ with $\epsilon_{dis}.aH = 0$	$aH' = aH + ev \cdot T$
Output relaxation constants: $\epsilon_{int}.aH$		

As an example for an action, *u2i* shown in table 20.1 expresses that the chassis level must be greater or equal to the lower bound plus some

constant ( $fH.val' \geq lb + c$ ), and that a reset event is to be emitted ( $reset!$ ).

States in HyCharts are labeled with *invariants*. Operationally, a transition exiting *can* be taken iff its action guard evaluates to true, and a HySChart *cannot* be in a state whose invariant evaluates to false, therefore ‘forcing’ it to take an outgoing transition. In order to avoid time deadlocks, invariants are required to be *sound*: for each state, there is always either an emerging transition which is enabled, or the invariant evaluates to true, or both.

Control states may be labeled with *activities* specifying the continuous part. Activities describe the continuous evolution of the component’s variables when control is in the respective state. Note that the time derivative of a variable  $v$  is written as  $\dot{v}$ .

In our example, activity  $a\_const$  associated with control states IN-BEND and IN-TOL specifies that variable  $aH$  remains constant. Note that the input/output relation  $Com$  resulting from the HySChart is required to be total for all states and inputs. This ensures that a HySChart cannot ‘reject’ certain inputs. Figure 20.1 shows in the middle column invariants and activities for the HySChart of CONTROL. The  $\epsilon$  values are introduced by the transformation rules explained in the ‘Relaxation’ paragraph below.

**Semantics.** The behavior of a HyChart is specified by the combined behavior of its components. Each of them is formally specified by the *hybrid machine model*. Figure 20.4, left, shows the machine model of a HySChart. It is constituted of a combinational part ( $Com^\dagger$ ), an analog part ( $Ana$ ), a feedback loop with an infinitely small delay ( $Lim_z$ ), and a projection ( $Out^\dagger$ ).

The feedback loop, combined with  $Lim_z$ , models the state of the machine. At each point in time  $t$ , the component can access the received input and the output ‘exactly before’  $t$ .  $Com^\dagger$  controls the analog part, and allows discrete manipulations of state variables, but does not have a memory. Depending on the current input and the fed back state,  $Com^\dagger$  computes the next state.

The analog part uses this next state to select an activity which specifies the continuous flow of the

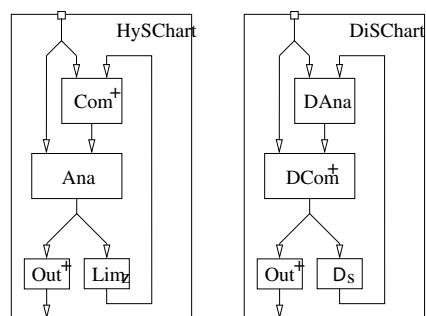


Figure 20.4. HyAChart of hybrid machine model

component's variables. The next state may also assert an initial value for the activity. Note that  $Com^\dagger$  may alter the component state only for isolated points in time. Between these points in time,  $Com^\dagger$  idles. For global composition of HyChart models, HyACharts specify the data flow between components, and HySCharts describe the hybrid machines.

**Time Refinement.** When discretizing HyCharts, two deviations from the original behavior are introduced: (1) due to sampling, transitions in the discretized model are taken some time after they would have been enabled in the idealized model, (2) sampling in the discretized analog part introduces *discretization errors* and *intersample errors*. Our notion of error refers to the deviation of the discretized signals from the original signals. Discretization errors at sampling instants, while intersample errors are deviations in between samples, assuming that the discretized signal remains constant.

For a tractable design flow, the notion of *behavior refinement* is helpful: A (more concrete) component  $A$  is a refinement of a (more abstract) component  $B$  if  $A$ 's input/output relation is contained in  $B$ 's input/output relation. As it preserves universal safety properties, this kind of refinement is also called *property refinement*. The property refinement considered here is a change of the underlying time model from mixed continuous/discrete to discrete. The (possibly) continuous evolution of a value in  $A$  is approximated by a series of discrete value changes in  $B$ .

**Relaxation.** While mathematically precise, the semantics of HyCharts or other non-relaxed hybrid formalisms is too "sharp" for discrete-time refinement. *Relaxed HyCharts* accomodate for these uncertainties in the system's behavior: A relaxed HyChart is constructed from a regular HyChart by relaxing its state invariants, analog dynamics, and outputs using a set of transformation rules. The relaxed HyChart therefore allows a larger set of behaviors.

Informally, the transformation proceeds along the following scheme:

**Construction of relaxed invariants.** In regular HySCharts, *state invariants* are implicitly associated with each control state. The invariant is constructed as the conjunction of the negated guards of outgoing transitions. This results in eager taking of the transitions; we therefore call such invariants *exact invariants*. The relaxation weakens the exact invariants so that some time passes between an action guard becoming true and the corresponding part of the invariant becoming false. We associate a relaxation constant  $\epsilon_x$  with each discrete and hybrid variable  $x$ . The HySChart may remain in the node for  $\epsilon_x$  time units after the last change of  $x$ . For con-

tinuous and hybrid values, threshold crossings are relaxed with a similar constant  $\epsilon_a$ , so transitions must be taken only if the value is significantly above the threshold.

**Relaxation of analog dynamics.** The relaxation introduces another relaxation constant: For each real valued variable controlled by an activity,  $\epsilon_{dis}.v$  specifies the allowed deviation of relaxed from non-relaxed behavior. For refinement, this relaxation accommodates for the discretization error.

**Relaxation of output variables.** The output relaxation is introduced by inserting an additional relaxation component at the HySChart's output interface. The relaxation component ensures that for each output variable  $x$ , all behaviors with a maximum deviation  $\epsilon_{int}.x$  are included in the HySChart's behaviors. Note that the relaxation may also introduce discontinuous jumps in the evolution of  $x$ . In our refinement technique, this relaxation accommodates for the intersample error.

Table 20.1, middle column, already shows the relaxed invariants and activities. A more detailed and formal treatment of the transformation rules can be found in [11].

**Discrete HyCharts.** *Discrete HyCharts* were developed for describing the discrete-time refinements of HyCharts models. Both regular and discrete HyCharts use HyACharts for structural description; for behavioral description, *DiSCharts* are introduced. Like HySCharts, DiSCharts are hierarchical, sequential control-flow graphs, but in contrast to HySCharts the underlying time model is discrete only. The DiSCharts machine model is similar to HySCharts (Figure 20.4, right): the infinitely small delay  $Lim_z$  is replaced by a unit delay  $\Delta_s$ , and the order of *Ana* and *Com* is reversed so that *Com* can immediately react to state changes caused by *Ana*.

**Construction of the DiSChart from HySChart.** Table 20.1, right column, shows the invariants and activities for the DiSChart of CONTROL (as the control flow is generally not modified when going from HySCharts to DiSCharts, the graphical representation is equivalent).  $T$  is the fixed sampling period for the discrete HyChart. The DiSChart was derived using a discretization method from [11]. In principle, other (fixed-step) discretization methods are possible.

The method makes the following assumptions on the chart's inputs and variables: (1) All variables which change continuously with time must be Lipschitz constrained with constant  $l$ , that is, there exists an  $l$

such that the time derivative is within  $\pm l$ . For the CONTROL example, we assume a Lipschitz constant  $fl$  for variable  $fH$ . (2) For all continuously changing variables, a maximum error  $e$  is given. In the example,  $fH$  has a maximum error of  $fe$ . (3) For each hybrid and discrete input channel, a minimum event separation  $m$  has to be provided.

It can then be shown [11] that the derived discrete HyChart of the EHC system is a discrete-time refinement of the relaxed HyChart if the sampling period  $T$  is chosen such that the following inequality holds:

$$T \leq \min\{\epsilon_{bend}, \frac{\epsilon_{i1} - 2 \cdot fe}{fl}, \epsilon_{fH}, \frac{\epsilon_{int} \cdot aH}{cp}, \frac{\epsilon_{int} \cdot aH}{|ev|}\}$$

With the discretized HyCharts model, we are now ready to introduce SystemC-AMS as the target format for the translation in the next section.

### 3. Modeling Hybrid Control Systems with SystemC

For the realization of hybrid control systems, continuous functions are realized by discrete hardware/software systems. The design of such systems starts with an executable specification, e.g. in Matlab/Simulink or SystemC. SystemC is a C++ library which supports the executable specification and design of complex, embedded hardware/software systems. In the following, we give a brief overview of means to model discrete systems in SystemC, and methods to model signal processing systems in SystemC(-AMS).

**Modeling and Simulation of Components.** In SystemC a component is an object of the class `sc_module`. Discrete behavior of components is specified by methods or threads. For example, we can model the behavior of a PI controller by a process as follows, provided the signal `clk` activates the process at constant time steps:

```
double state, k_i, k_p;
class pi_controller: public sc_module
{
    sc_in<double> input; // in port
    sc_in<bool> clk;    // clock
    void do_step
    {
        state += k_i*input.read();
        output.write(state+k_p*input.read());
    };
};

SC_CTOR(pi_controller)
{
```

```

    SC_METHOD(do_step) sensitive << clk;
    // executes do_step at each event on clk
}
}

```

Of course, the continuous behavior of analog components such as the above integrator is only approximated. In the same way as above, one can model components used in section 20.4, such as unit delay, multiplexer or arithmetic functions.

**Modeling and Simulation of Block Diagrams.** If a number of blocks with discrete or continuous behavior, that communicate is combined to an architecture we must introduce a model for communication and synchronization. In SystemC, the communication between processes respectively modules is modeled by channels (signals). For example, we can specify a structure, where the controller gets an input from a system **s1** as follows:

```

sc_signal<double> a, clk; // channels used
pi_controller ctrl1(); // pi controller module
system s1(); // yet another module
ctrl1.input(a); ctrl1.clk(clk); // connect modules
s1.output(a); s1.clk(clk);

```

In the above example the method **ctrl1.input(a)** is called *before* simulation starts. The method notifies the simulation manager about a connection between an abstract port (which is only an interface) and a signal, which realizes this interface.

For system level design of signal processing systems the static dataflow model of computation as introduced in section 20.1 is more appropriate than discrete event simulation. It is actually implemented in prototypes of SystemC-AMS ([13]).

In SystemC-AMS clusters of signal processing blocks are controlled by a cluster manager or coordinator as shown in figure 20.5. The coordinator determines a schedule of the modules in the data flow's direction using the information of ports and directed signals. Then the coordinator simulates all blocks in constant time steps in this order. For example, in figure 20.5, for a given input of the SystemC kernel and a known delayed value, the coordinator would first ask block *sigproc1* to compute his outputs, then *sigproc2*, and finally *sigproc3*. After a delay the same procedure would start again, and so on.

In the following, we assume a realization in SystemC-AMS. For visualization, we use the Simulink block diagram editor.

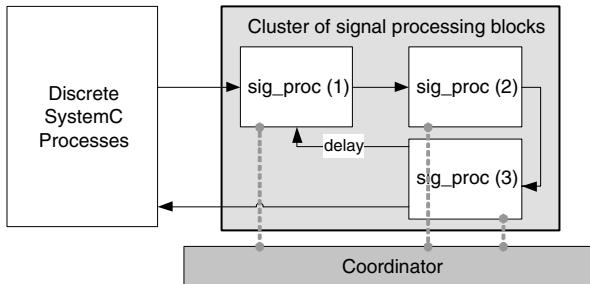


Figure 20.5. SystemC-AMS simulation of a signal processing cluster

#### 4. Translation of discrete HyCharts to SystemC

In this section we show how a discrete HyChart can be translated into an equivalent composition of SystemC blocks. For the current lack of a suitable block diagram tool in the SystemC context, we chose Simulink as a block diagram editor in order to demonstrate the translation. We assume some kind standard block library exists with a set of *arithmetic*, *boolean comparison*, *unit delay*, and *signal routing* blocks. Signal routing includes *multiplex* and *demultiplex*, *switch* (given some  $n$  and a vector  $\vec{v}$  as input, output the  $n$ -th component of  $\vec{v}$ ), and *selector* (given some vector  $\vec{v}$  as input, output a vector  $\vec{v}'$  with some of  $\vec{v}$ 's components) blocks. We furthermore assume that simple expressions as in the discrete HyChart's actions and activities are directly encoded as SystemC-AMS blocks.

The discrete HyChart has to meet the following criteria in order to be translatable: (1) it is deterministic, i.e. its internal state and its outputs are completely determined given an input, (2) there are no delayless loops in the component network, (3) the identification connector  $\bullet\llcorner$  is not used in HyACharts (identification has not been used in any HyChart model so far). (4) it is total on all inputs and states, hence inputs cannot be rejected. The latter condition is always fulfilled by our refinement procedure, as totality of the discrete *Com* part's input/output relation follows directly from totality of the original (non-relaxed) HyChart's *Com* part. Each HyChart operates on a data state  $\sigma \in \mathcal{S}$ , an input  $\iota \in \mathcal{I}$ , and a control state  $\kappa \in \mathbb{N}$ .  $\mathcal{S}$  is composed of the product of the types of all variables accessed in the HyChart's activities. The control state  $\kappa$  is some natural-number encoding of the HyChart's control states. Extension of the encoding to hierarchical states by shifting namespaces is described in [12].

**Translation of multiplicative and additive graphs.** As a useful prerequisite for the translation, the multiplicative (parallel) composition operators in HyACharts naturally reflect composition in dataflow-oriented design languages like Simulink or SCADE/Lustre [7]. We therefore do not show an explicit example for HyAChart translation, and concentrate on HySCharts instead. The additive (sequential) composition of  $m$  nodes in HySCharts is generally translated as  $m$  parallel blocks, with a switch block choosing the appropriate output according to the current control state. We'll see examples for the translation below.

**Machine model.** The top-level translation of a HySChart follows directly from the discrete-time hybrid machine model sketched in Figure 20.4, right: The discrete-time variation of the analog part, *Ana*, computes the next state ( $\sigma.fH(kT)$ ,  $\sigma.reset(kT)$ ,  $\sigma.bend(kT)$ ) from the current inputs ( $\iota.aH(kT)$ ,  $\iota.bend(kT)$ ), the latched control state  $\kappa((k-1)T)$ , and the latched data state ( $\sigma.fH((k-1)T)$ ,  $\sigma.reset((k-1)T)$ ,  $\sigma.bend((k-1)T)$ ). The combinatorial part *Com* then computes the next control and data state from the current input and *Ana*'s outputs. The chart's output  $o \in \mathcal{O}$  is computed by projecting the data state onto  $\mathcal{O}$  using a selector block (Figure 20.6). The unit delay in the feedback loop finally is realized with unit delay ( $\frac{1}{z}$ ) blocks.

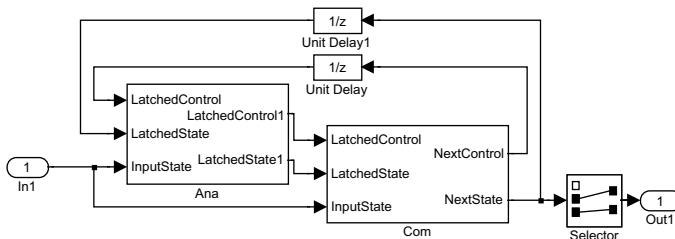


Figure 20.6. Block diagram translation for *Control*

**Analog part.** Specification of the discrete evolution law is typically done with difference equations, such as the activities in table 20.1. In the translation, each single activity in the discrete HyChart corresponds to a SystemC-AMS block computing the corresponding difference equation. The currently active activity is then selected by an  $m$ -ary switch block based on the current control state. Note that the control state is not controlled by *Ana* and simply fed through. Figure 20.7 illustrates the translation.

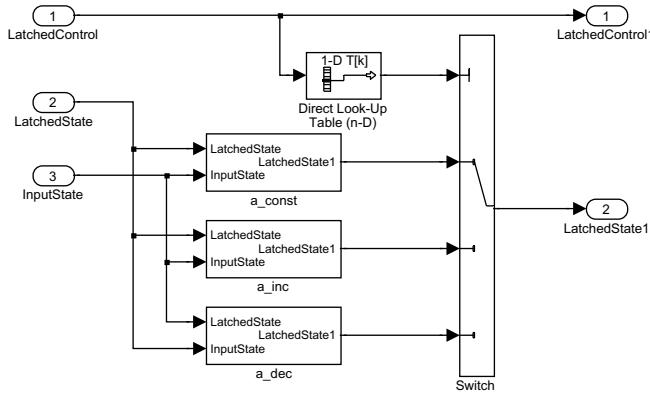


Figure 20.7. Block diagram translation for *Ana*

**Combinational part.** In HyCharts, the semantics of the combinational part are given in terms of a syntactical transformation of primitive control states (*additive nodes*) to hierachic graphs. Each node of the hierachic graph is entered through one of its  $m$  entry points, executing their entry actions, or through an explicit wait entry point if there is no change in control state. The appropriate action is chosen by evaluating the *guards* associated with each action. Hierarchical control states are transformed in a similar manner. In our translation, the  $m$  action guards are translated to  $m$  parallel SystemC-AMS blocks which compute the guards. The next control state is then chosen using, for instance, an  $m$ -ary *look-up table block*; determinism of the chart ensures that only one guard at a time evaluates to true. If none of the guards evaluates to true, the chart remains in the same control state, corresponding to the invariant guard in the hierachic graph. Figure 20.8 shows the translation for the EHC system's *Com* block. Note that the control state hierarchy has been flattened so that INBEND, UP, INTOL and DOWN are on the same level of hierarchy.

**Actions and activities.** In this translation, evaluation of assignments, arithmetic expressions, and boolean predicates in actions and activities is handled at the block level. In SystemC-AMS, such expressions are simply encoded as equivalent C language statements within blocks. Note that in the C language translation, the term  $now \bmod T \neq 0$  is removed from each of the DiSChart's invariants as the simulator enforces discrete steps anyway.

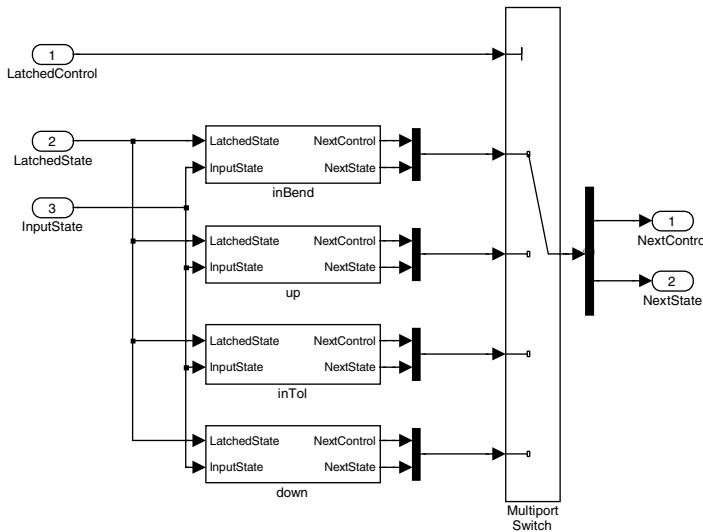


Figure 20.8. Block diagram translation for *Com*

As mentioned above, extension of the above translation to parallel composition of discrete HyCharts is straightforward as multiplicative composition in HyACharts is naturally represented in block diagrams.

## 5. Conclusion and Future Work

We have presented an integrated approach for the specification and refinement of hybrid systems using a formal, graphical design notation to capture abstract designs of hybrid systems. Based on a relaxed version of the model, the design is discretized and translated to a model in a design language. Correctness of the translation is ensured by evaluating a set of constraints, ensuring that the discretized model is a time refinement of the relaxed model. In principle, the above approach may be extended towards other discrete-time languages used for synthesis of HW/SW systems. Synchronous languages like Lustre [7] or AutoFOCUS [8] would be natural targets for our translation.

**Acknowledgements.** Both authors would like to thank Thomas Stauner for commenting on a draft version of this manuscript.

## References

- [1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic

- analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
- [2] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement of hierarchical hybrid systems. In *Proceedings of HSCC 2001*, LNCS 2034. Springer Verlag, 2001.
  - [3] M. Broy. Refinement of time. In Th. Rus M. Bertran, editor, *Transformation-Based Reactive System Development. ARTS'97*, Lecture Notes in Computer Science 1231, pages 44–63, 1997.
  - [4] D. Gajski, F. Vahid, and S. Narayan. A System-Design Methodology: Executable-Specification Refinement. In *The European Design Automation Conference (EURO-DAC'94)*, Paris, France, February 1994. IEEE Computer Society Press.
  - [5] Ch. Grimm. Modeling and Refinement of Mixed Signal Systems with SystemC. In *Methodologies and Applications*, Boston/London/Dordrecht, 2003. Kluwer Academic Publishers.
  - [6] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proc. of FTRTFT'98*, LNCS 1486. Springer Verlag, 1998.
  - [7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
  - [8] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.
  - [9] P.J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Proceedings of HSCC 1999*, LNCS 2034, pages 165–177. Springer Verlag, 1999.
  - [10] B. Selic, G. Gullekson, and T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
  - [11] T. Stauner. *Systematic development of hybrid systems*. PhD thesis, Technische Universität München, 2001.
  - [12] T. Stauner and C. Grimm. Übersetzung von HyCharts in HDFG. In *ITG/GI/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Berlin, Germany, 2000. VDE-Verlag.
  - [13] A. Vachoux, Ch. Grimm, and K. Einwich. SystemC-AMS Requirements, Design Objectives and Rationale. In *Proceedings of Design, Automation and Test in Europe (DATE'03)*, 2003.

# V

## APPLICATIONS AND NEW LANGUAGES

The following chapters give an overview of recent developments. The first paper describes upcoming requirements to new languages and design methods from the point of view of automotive software engineering. In this domain, the software parts will become more significant, and software engineering will become more important. The second paper gives an overview of SystemVerilog and ongoing work to its standardization. SystemVerilog augments the hardware description language Verilog with new features known from C-Based languages such as SystemC, and other features that improve test and synthesis.

Christoph Grimm

*J. W. Goethe-Universität Frankfurt, Germany  
grimm@ti.informatik.uni-frankfurt.de*

*This page intentionally left blank*

# Chapter 21

## AUTOMOTIVE SOFTWARE ENGINEERING

*An emerging application domain for software engineering*

Christian Salzmann, Thomas Stauner

*BMW Car IT GmbH Munich*

**Abstract** As the innovations in the automotive field shift more and more from mechanical solutions to electronic functions, the automotive domain emerges to one of the most potential application domains of software engineering. In this paper we explain the differences between automotive software and the classical embedded software domain and sketch out what the main application fields for software research in the automotive domain are.

**Keywords:** Software Engineering, Automotive, Methodology, Domain Specific Architectures, Embedded Systems, Software Architecture, Business Aspects.

### 1. Introduction

Modeling and the development of software differs heavily between different domains: it seems obvious that the type of software used in embedded systems like an airbag control unit differs from those used in business information systems or even web based systems. But not only the type of software, e.g. the used programming language, the operating system or the platform, are domain specific, but also the requirements, methods and techniques for developing software.

In this paper we plead for a specific domain of software development, *automotive software engineering* (ASE) and present the reasons why.

The automotive industry faces a paradigm shift from mechanical innovations to software innovations: in the last years the innovations in new models were driven by electronic functions. By increasing the amount

of electronic functions the software emerges as a central aspect in the car. In the current BMW 7 series for example we face a total amount of 270 functions the user interacts with, deployed over 67 independent embedded platforms which sums up to about 65 megabytes of binary code.

**The Past.** The software in the car used to be isolated, small programs that run inside an embedded device. The embedded devices acted as single nodes in the distributed software system of the automobile. Usually the embedded code used to be in the area of some kilobytes per device, a car used roughly a dozen devices. The usual abstractions were minimal and the focus was on lean resources. Therefore usual platforms were minimal software platforms and applications were designed directly in machine code or in close languages such as C.

**The Future.** However, the amount of software changes rapidly. Cars of the next generation, which are expected within the next six years, will reach the amount of one gigabyte of binary code. This is comparable to a state of the art desktop workstation of today. It seems obvious that a system of that size cannot be developed with the methods and abstractions of classical embedded software. But a workstation and its software faces completely different needs as the software of a car faces: Product lifetime of a car is at least 10 years vs. 3 years of an office product (with several hot fixes), the strong time-to market-pressure of an office feature vs. the demands for reliability of automotive software etc. We will therefore need similar levels of abstractions, as we know from the desktop world, however tailored for the automotive domain.

**Overview.** In the following section we are analyzing the characteristics of automotive software concerning not only technical aspects, but also methodology and organizational aspects. After defining the characteristics we reason our demand for automotive software engineering and point out what the main application fields for software engineering research in the automotive domain are.

## **2. Characteristics of Automotive Software Engineering**

What differentiates automotive software and the way it has to be developed from other domains, like business software or aerospace for example? We first want to sketch out some main observable symptoms of automotive software before starting to analyze them and breaking them down to a few main characteristics. The different types of symptoms of automotive software, which include besides technical issues also organi-

zational and economical issues, may also illustrate the broad variety of challenges that go far beyond what one might expect of the development of automotive software as classical embedded code.

## 2.1 Observable Symptoms

**Integration of Different Software Domains.** One may think that automotive software is exclusively highly mission critical software, as the often-cited airbag software or the drive-by-wire systems. Those demand for a maximum of reliability and correctness. However, by analyzing the software of premium cars you can identify the whole range of software from the mission critical airbag or steer-by wire software up to infotainment software:

- *Automotive Client Systems:* All client software that interacts with the vehicle itself, such as mobile phones, PDAs via Bluetooth and Laptop computers.
- *Human Vehicle Interaction:* This is the central interaction system between the user and all electronic functions (e.g. infotainment, climate control, seat positions etc.) of the car.
- *Integrated Data Management* Integrated data management systems that manage data centrally (not locally on each ECU) and establish a data infrastructure in the car.
- *Software Transfer Software* must be integrated into the vehicle infrastructure. All management of software coming into a productive automotive system as well as the transfer itself are classified into this area.
- *Automotive Control Systems:* All mission critical software and systems where a malfunction could harm human beings are classified with very high safety constraints and must feature hard real-time capabilities.
- *Automotive Comfort Systems.* This includes further control systems and reactive systems which may have real-time constraints, but which are not safety relevant. Examples are window opening and interior light control.

**Close Interleaving Between HW and SW in the Development.** Usually software is mostly developed independently from its hardware platform. This is done for good reasons: giving more flexibility for deployment, higher compatibility with future hardware generations etc. As a matter

of fact, automotive software, however, is closely developed *together* with its latter hardware platform, the ECU (Electronic Circuit Units), since the software is optimized to the hardware for performance and resource allocation. So we have a close interrelation between the software development and the hardware development process. The reason for this is the fact that software is tightly optimized for the concrete hardware platform.

The reason for this lies in the fact of unit based cost structures (see below) and therefore the software gets optimized for the according hardware and vice versa.

**Emphasis on Integration of Independently Developed Software.** In the current BMW 7 series there are up to 67 different platforms. Most of these platforms are specified by the automotive company, i.e. the OEM (Original Equipment Manufacturer), and independently developed by different suppliers and subcontractors. Only a small amount of strategically software is specified, designed and developed by the OEM itself. So automotive software engineering needs to have a strong emphasis of *integration* of independently developed software. Software that has to be integrated and comes from a different vendor may only be specified by its black box behavior or by its interfaces. Internal mechanisms may be unknown, and even the specification itself may be wrong or largely not equivalent with its implementation. This implies concrete and tailored techniques that help to manage all problems that are known from software integration, such as feature interaction.

**Safety and Security.** Safety and security takes a crucial role in the automotive domain. It is easy to see that especially safety concerns are important for all driving relevant functions, like drive by wire or passenger safety functions. However, with the developing infotainment functionality the car emerges to an information hub where functions of cell phones (UMTS, Bluetooth), Laptops (Wireless LAN) and PDAs are interconnected via and with the car information systems. Therefore personalization and the related security issues are getting important.

The goal of ASE must be to differentiate between the various software domains in the car (i.e. infotainment, driving functionality, etc.) and offer the proper safety and security techniques, both for the software infrastructure and the development process.

**Lifecycle Gap Between Software and Car.** Software has a typical lifecycle of about 2-4 years. However, a car model is typically produced for about 6 years. Obviously we cannot produce cars with a 4 year old

software, on the other side one can not shorten the lifecycle of a complex product such as a car. How can we find mechanisms to bridge this gap? Software updates alone may not help since the update of software may bring new potential incompatibilities as well as it may imply the according hardware update.

**Variants.** A premium car typically has about 80 electronic fittings that can be ordered depending on the country, etc. This gives us roughly  $2^{80}$  variants a car can be ordered and must be produced. Additional to this, each piece of software in the car has various versions. This is the reason why hardly all cars in the premium segment are more or less unique and we have a huge amount of variants of the automotive software that must be handled technically and economically.

Technically the software of all variants must be tested and integrated with all other variants which makes (using the above cited numbers)  $2^{80}$  configurations that have to be tested. Obviously, an elaborate design and test methodology is required for this.

Economically it makes much sense to reuse large parts of the software and even produce the software in a way that it is future-proof for coming new variations. Software-Product-Lines (SPL), which develop certain artifacts for reuse from product to product, may be a promising way to go.

**Unit based Cost Structure.** For all decisions on how car functions are realized, the cost per produced unit plays the decisive role. Due to the large produced quantities, production and material cost by far outweigh engineering cost. An ECU may be produced over 6 years or more with e.g. 500,000 units per year. Thus, 3 Mio \$ can be saved over the production period, if the cost per unit can be reduced by 1\$. This explains that a lot of engineering effort is typically spent on optimisation to keep the cost per unit low.

For the future we conject that time to market and the ability to develop highly reliable complex systems will increase in importance and sometimes justify higher unit cost.

**Collaboration based, Distributed Development Process.** Development is usually conducted in a collaboration of OEM (Original Equipment Manufacturer) and suppliers. This has the primary advantage that specialized know-how at the suppliers site can be used. The supplier can use synergies in development and production, since he usually produces similar systems for different OEMs, which also keeps unit cost low for the OEM. For functions that do not differentiate the different OEMs

this synergy is exploited particularly strongly. As a negative side effect development is geographically distributed and communication gets more complicated this way.

## 2.2 Main Characteristics of ASE

All the above described and observed symptoms that make the development of automotive software different from other domains are from our perspective based on three main axioms:

**Heterogeneity:** As already explained in the introduction, automotive software is very diverse ranging from entertainment and office related software to safety-critical real-time control software. For the rest of this paper we cluster automotive software into three groups: *Infotainment & Comfort Software* (including system services with soft real-time requirements), *Safety Critical Software* and (*hard*) *Real-time Software*.

**Emphasis on Software Integration:** As a result of the collaboration-based development process OEMs have to integrate independently developed systems or software. While this always is a difficult task for a complex system the situation is automotive software engineering is even worse, because suppliers usually have a lot of freedom in how they realize a solution. (In business IT the client would often strongly constrain the technologies that may be used by a supplier to facilitate integration and maintenance.) Furthermore, the OEM usually only has black-box specifications of the subsystems to be integrated which makes successful testing and error localization more difficult. Besides that distributed development causes that it is difficult or impossible for the OEM to modify parts of the subsystems in order to localize errors.

**Unit based cost structure:** The unit based cost structure has already been mentioned as a business characteristic above. For automotive software engineering it is problematic, because it drives system engineers to design hardware that is as cheap as possible for the required function. This usually means that performance and memory is small. Thus, the software engineer is confronted with a hardware that is only sufficient for the function to be implemented, if the implementation is highly optimized. In particular this means that abstraction layers possibly have to be sacrificed to performance and modern development methodologies and programming languages often cannot be used. This of course makes developing complex software that is correct and delivered in time a very difficult task. In the recent past important automotive system development projects were late because of software problems.

### 3. The Demands for an Automotive Software Engineering Discipline

Due to the software gaining more and more importance in the automobile and due to the increasing complexity of its application the demand emerges for a tailored software engineering discipline for automotive software. This software engineering discipline, like other disciplines for important application fields like business software, multi-media applications or telematics, should focus on the problems and characteristics of the domain (i.e. automotive) and help to solve the existing problems. Such an automotive software engineering discipline would need to cover all relevant areas of software engineering and present tailored solutions for the specific needs of the emerging market. In this paper we will only sketch the development *process* for automotive software.

As described above, automotive software is characterized by heterogeneity of the software types, the integration of independently developed software (OEM and suppliers) and the unit based cost structure. An automotive software development process must support these three key characteristics. In the early phases different software types like infotainment software and a real-time control SW must be developed differently, concerning the abstractions and the techniques. However in the latter phases when it comes to integration and system tests these differently developed software units must be integrated and therefore the various documents and descriptions of the software must be set in relation.

We therefore need (1) a development process that splits in the beginning into several development techniques and (2) abstractions that fit each software type. In the latter phases the descriptions must be able to be set in relation.

We therefore think that a model-based approach is a promising one: a base model with several views and abstraction levels for each software domain.

In the following we outline for each process phase typical activities in automotive software development. For all of these activities supporting techniques and methods are still needed.

#### 3.1 Process Paradigm

For the overall process paradigm we propose an iterative, incremental process. The reason is (1) that in vehicle development customer requirements are often unclear and change during the development process and (2) for some product properties feasibility studies are necessary to validate whether the requested feature can be implemented with the desired

quality and cost. This can also motivate the making of throw-away prototypes instead of incremental development.

According to the iterative process paradigm all the phases listed below should be applied in several iterations.

Due to the heterogeneity of automotive software, aspects of the overall process paradigm become more important or less important depending on the characteristics of the domain:

- *Infotainment Applications* (including soft real-time applications). For infotainment applications throwaway prototypes are particularly important to verify performance requirements early in the development process. In later steps it may be difficult to modify the HW or the software architecture.
- *Hard real-time Applications*. For hard real time applications (and also for safety relevant systems) some prototypes should be replaced by analytic methods and models to verify certain properties, e.g. WCET (worst case execution time) analysis to prove that certain deadlines can always be met.
- *Safety relevant systems*. For safety relevant systems all process phases must be accompanied by activities that assess the safety impact of all decisions and that verify that the process phase considers all safety-related requirements that must be regarded in it. E.g. for the partitioning of functions on ECUs, the requirement that a single HW fault must not lead to a safety hazard has to be considered.

Note that accompanying processes like project management, configuration management and supplier agreement management are not considered in this paper.

## 3.2 Requirements Engineering

Due to the heterogeneity of automotive software we need suitable requirement engineering techniques for each domain. The best practice approach so far is the following.

- *Infotainment Applications* are characterized by very complex, multimodal user interfaces, which makes it hard to specify the requirements formally. Furthermore a “good” usability is not specifiable, but must be discovered by experiments. Therefore a high-level verbose description of top-level requirements is the best practice.
- *Real time Applications* are mostly embedded functions with clear functionality from the beginning of the development process. For

control systems large parts of the requirements follow from the kind of function they must realize (e.g. stabilization) and from the underlying physics. Therefore it is possible and desirable to have an (abstract) black box description of the interface of the system including the messages and the black box behavior of the system.

- *Safety relevant systems* have especially hard availability, correctness and robustness requirements. The concrete safety requirements for a system are usually determined from a functional hazard assessment. Similarly for security requirements threat analysis is used. Black-box interface descriptions of the system can also often be used for safety critical systems.

For real-time and safety critical applications an explicit environment model should be created at this point of the development process. It is refined in later phases for simulation and verification purposes.

For infotainment applications a usual domain model suffices.

### 3.3 Software Architecture & Design

Software architecture plays a key role in the automotive IT domain. As described above the development process is collaboration based: the OEM specifies the software, a supplier develops the software or the system, based on the specification and returns a "black box" of software to the OEM. The OEM later on has the job to integrate and test all the differently developed software parts, based only on the existing specifications. We need the interfaces, the principle interaction mechanisms, their communication protocols and structure, which boils down to a model of the software architecture [11]. Key demands for the automotive domain are:

- **Layered Architectures:** With the increasing amount of software we need abstraction layers for communication infrastructure and distributed processing. As we see such abstraction layers in the desktop domain, such as TCP/IP protocols for communication and CORBA for distributed processing we need tailored abstractions that fit the specific needs of the automotive domain. The AUTOSAR standard [3] for distributed processing, which is currently under development, represents an example for an automotive approach for layered architecture.
- **Open, Modular Systems:** Today's automotive software is mostly optimized for the hardware to lower the unit costs of the ECU. To be able to reuse software a modular software architecture is needed

that allows us to model software components as reusable software units as we know it from the business IT. Nevertheless this component model must be tailored for the automotive needs, like resource allocation, embedded environments etc. First approaches towards an modular software architecture were taken in projects like EAST [6] and Cartronic [7]. A new promising candidate is the AUTOSAR consortium, where reusable automotive software components are standardized.

- **Interface and Communication description:** To be able to integrate and especially to test the integrated system new interface descriptions are needed that describe the relevant attributes of the component including signature of the interface (signals), timed black box behavior (protocol), communication constraints (bus protocols etc.) and resource constraints (memory allocation etc). The goal must be to describe the software units in a way that allows the OEMs to process a complete software integration test, including feature interaction tracking, without knowing all the internals of a software components. This is certainly one of the most challenging issues in the future. An interesting approach may be [1].

In the development process the goal of the architecture phase is to create a platform independent, logical architecture of the functions in the vehicle. Before the transition to the specification phase the logical architecture is mapped on the physical architecture in a first iteration. This implies that a partitioning of functions in HW and SW and of software-implemented functions on processing units must be defined.

In architecture design it is important to map functional requirements on architectural entities, thus these requirements are "discharged" and the remaining requirements have to be considered in later phases. Such a clear link between requirements and architectural entities can help to discover feature interaction problems resulting from conflicting/missing requirements or causal loops between environment and system elements. Unfortunately, there is a deficit of methodological support for this activity.

- *Infotainment Applications.* For infotainment applications besides functional requirements quality of service attributes must be taken into account.
- *Hard real-time applications.* For these applications timing constraints must be identified in the requirements and associated with the respective architectural entities. However, at this level of gran-

ularity only timing constraints that are enforced by the environment or the requirements should be considered. Technical aspects, like scheduling, are considered in later phases.

- *Safety relevant systems.* For safety relevant systems we have to surveil that all requirements were considered appropriately (i.e. either mapped on architectural entities or deferred to later phases). Furthermore, a (preliminary) system safety assessment should be conducted to validate that the propagation of faults satisfies the safety requirements.

### 3.4 Specification

In the specification phase the logical functions are detailed. A functional decomposition of the architecture is performed and abstract interface specifications are concretized by glass-box models. At the point of functional decomposition the mapping of logical functions on processing units will often be revised, in particular in the context of logical functions that are distributed among several computation units. A frequent reason for these changes in the mapping is that the lower level architecture resulting from the decomposition may reveal an unnecessarily high amount of bus traffic between ECUs. Tool support for such an analysis is highly desirable.

Since the specification phase already has to deal with ECUs, lower level software layers of the ECUs, like those implementing diagnosis or flash programming, also have to be considered in this phase.

Specializations for the vehicle domains are as follows:

- *Infotainment Applications.* For infotainment applications informal text (together with quality of service specifications) will usually be used instead of glass-box models defining the functionality.
- *Hard real-time applications.* For these applications the glass-box models must be complemented by specifications of the scheduling strategies. This means that a meta model is needed here which allows the unambiguous integration of information on different abstraction layers. Furthermore, in this domain custom HW will sometimes be advantageous. Therefore a HW/SW codesign paradigm can also be employed here in order to decide which models are implemented in SW and which in HW.
- *Safety relevant systems.* For safety-relevant systems simulation and formal verification will often be used extensively to verify that safety requirements are satisfied, in particular if the regarded

function is not used often during normal vehicle operation. Furthermore, for this system domain integrity requirements are now mapped on architectures with redundancy (in particular HW redundancy). Redundancy can probably be considered as an aspect that is interwoven with normal system functionality: Standard architectures like, triple modular redundancy, are used to compose several functions with the same functionality.

### 3.5 Implementation

Also the implementation phase differs from software type to software type. When classical embedded software was implemented directly in assembler or C automotive infotainment applications are implemented with object-oriented abstractions in C++ or even with type safe languages like Java.

- *Infotainment Applications.* For infotainment applications high level languages are already in use in current premium cars. The high availability of developers and the quality of the code together with low requirements for resource efficiency make Java or C# promising candidates.
- *Hard real-time applications.* In this domain, where hardware-software codesign is important, C is the state of the art, possibly generated from fine grained models like those supported by the ASCET tool.
- *Safety relevant systems.* For safety-relevant systems reactive languages like Esterel [4] might be promising candidates for the future.

### 3.6 Test

Before digging deeper into the test phase it is important to mention that validation and verification are activities that have to be performed as a part of every process phase. Furthermore, we note that test case specifications have to be produced as a result of every preceding process phase, because a test always requires that a correctness criterion is defined. Besides that, test interfaces must have been included in the preceding phase to enable effective testing. For instance, if a system cannot be forced into a specific state by the test environment this state cannot be tested sufficiently (cf. Design to Test).

The test process itself consists of several sub phases (as seen from the SW view): SW unit test, SW integration test, HW/SW integration test, subsystem integration test and system integration test. For each of these subphases different technical problems have to be solved to enable

testing. Besides the importance of these problems we want to note that for systematic testing it is even more important to know what to test. In general we expect test case generation from specifications accompanied by random, risk-based testing to become state of the art.

- *Infotainment Applications.* For infotainment applications we see at least two further problems that have to be tackled. First, test cases for QoS requirements must be derived. There are not enough generation-based techniques for this problem yet. Second, in HW/SW integration and the later subphases as well the system input and output is difficult to stimulate/analyse, since it consists of audio or video output and input signals from multi purpose input units such as the BMW Group's I-Drive system, comparable in its functionality to a computer mouse.
- *Hard real-time applications.* For these applications special attention has to be paid to verifying that timing constraints are met under all conditions. This can require to generate load and memory consumption during a test.
- *Safety relevant systems.* For safety relevant systems manual design steps should be automated as far as possible. This means that code generation should usually be used, which shifts testing of the implementation process to testing of generator and compiler correctness. For these systems it is furthermore important to document tests for reviews by certification authorities.

## 3.7 Maintenance

Maintenance is influenced by two facts: 1) the maintenance personnel are mostly mechanics, meaning not software experts. The software maintenance must therefore be plug-and-play. 2) The maintenance costs must be a part in the total cost model of the software. Otherwise flexible software maintenance will not be possible since the software will always be tangled with its hardware and basically not maintainable.

- *Infotainment Applications* are characterized by relatively short product cycles: the look & feel of the UI must be compliant to actual taste, new services must be integratable etc. Updates must be easy, remote updates as known from desktop systems may be possible. Due to the frequency of updates openness of the system and the general resource abilities must be taken into account of the cost model of the hard- and software.

- *Hard real-time applications*: For hard real-time applications maintenance tools must not be accessible for the customer. Only expert personnel must be able to access and maintain the functions.
- *Safety relevant systems*: should be developed as a closed system with no or little variability for maintenance. If a safety relevant system, like an active front steering, must be changed, it should be replaced with a completely new developed system. If the system is not updated as a whole, a thorough analysis together with auxiliary verification activities is necessary to prove that all changes still result in a safe system.

## 4. Conclusion

In this paper we substantiated the claim that automotive software engineering is a specific domain of software engineering. To do so we elaborated essential characteristics from a number of observable symptoms of automotive software development. These characteristics are the heterogeneity of the systems, the integration (done by the OEM) of software that is developed independently by the suppliers, and the unit based cost structure.

Along a rudimentary development process, we pointed out key elements in automotive software development processes and hinted at directions of future research. In essence we think that ASE can greatly benefit from domain specific development techniques with clearly defined interfaces between them and with techniques for the efficient integration of the resulting software.

## References

- [1] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of the 9th Annual Symposium on Foundations of Software Engineering (FSE)*, ACM Press, 2001.
- [2] A. Aho, N. Gallagher, N. Griffeth, C. Schell, D. Swayne. SCF3/Sculptor with Chisel. In *Proc. 5th Feature Interactions in Telecommunications and SW Systems*. IOS Press, 1998.
- [3] Autosar - Automotive Open System Architecture. <http://www.autosar.org/>, 2003.
- [4] G. Berry. The Foundations of Esterel. In *Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, Foundations of Computing Series, 2000.

- [5] Manfred Broy. Automotive Software Engineering, In *Proceedings 25th Intl. Conference on Software Engineering*, 2003, Portland, Oregon, S. 719 ff.
- [6] EAST-EEA - Embedded Electronic Architecture. <http://www.east-eea.net/>, 2003.
- [7] H. Hülser, K. Benninger, J. Gerhardt and H. J. Glass. CARTRONIC - The ordering concept for flexible configuration of electronic vehicle subsystems. In *Proc. of AVL-Congress "Engine and Environment"*, Graz. 1998.
- [8] D. L. Parnas. On the criteria to be used in decomposing a system into modules. *Communications of the ACM* 1972 Vol. 15 Nr. 12 (December) p. 1053 - 8.
- [9] Alexandre Saad. Software im Automobil - Ausgangslage, Zielsetzung und Aufgabe der BMW Car IT. *Automotive Electronics*, 1/2003, S. 22 - 26
- [10] Alexandre Saad. Freude am Fahren - Prototyping bei der BMW Car IT GmbH. *JavaSpektrum*, 03/2003, S. 49 - 52
- [11] Mary Shaw and David Garlan. *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall 1996.
- [12] Clemens Szyperski. *Component Software*. Addison Wesley, 1999.
- [13] Pamela Zave. *Systematic Design of Call Coverage Features*. Technical report AT&T Labs 1999.

*This page intentionally left blank*

# Chapter 22

## SYSTEMVERILOG

Wolfgang Ecker<sup>1</sup>, Peter Jensen<sup>2</sup>, Thomas Kruse<sup>1</sup>

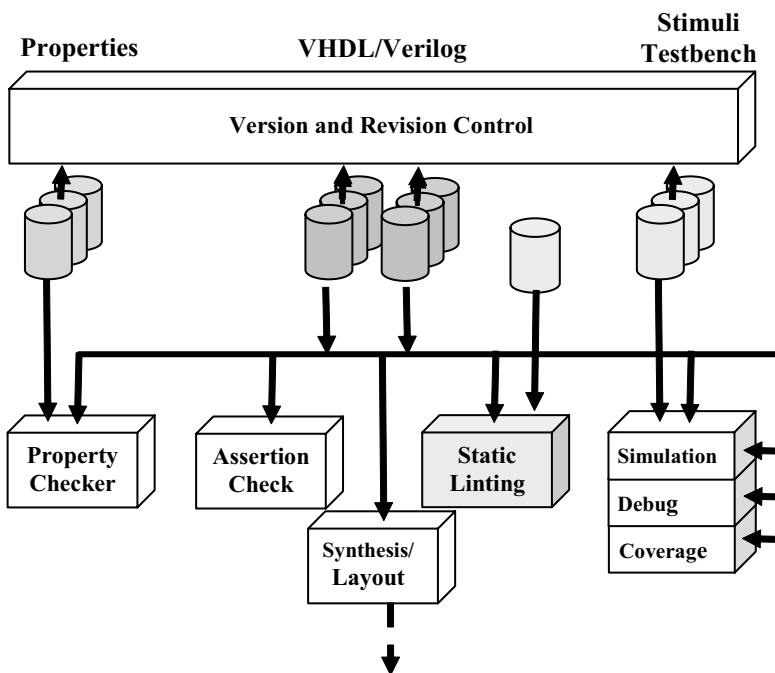
<sup>1</sup>*Infineon Technologies*

<sup>2</sup>*Syosil Consulting*

**Abstract** This paper gives a snapshot on the current activities in the Accellera SystemVerilog standardization. Needs and requirements from the user's side are covered as well as major new concepts and current status of the standardization and of available tools. The paper also gives links to papers currently available on the web.

### 1. Introduction

The design and verification gap is a world wide accepted challenge in the current semiconductor and microelectronic area. Re-use of existing design blocks and increase use of software executed by programmable CPU cores and DSP cores seems to be appropriate strategies to cope with the gap in design productivity. Design strategies as SoC, IP-based design, or platform-based design rely in one or the other way on re-use. In parallel, abstract models are made for specification validation and architecture analysis to get early feedback on the selected architecture and to prevent from long error correction cycles from gate-level models or RTL models to the specification. As reported in many talks and papers, verification consumes around 70% of the overall design time. For that purpose, a set of tools have been developed, which extend the pure HDL based design and verification flow. For simplification reasons, we focus in the rest of the paper to software-based verification tools only and omit hardware-based tools such as accelerators, emulators, or FPGA prototyping tools. An overview is given in Figure 1. Just as reminder, HDL Simulation and Debug were the only verification strategy used at the early days of broad use of HDLs, this means around 10 years ago.



*Figure 22.1. Design data and tools in a modern verification flow*

Both, design and testbench were written in one HDL and executed on the same simulator. Sometimes, generated stimuli were used also.

So-called static verification tools are introduced nowadays. These tools validate the model but do not execute it. The static linting analyses the model without consideration of functional aspects. General coding rules and design style rules are checked. A violation of such a rule shows a potential source of a bug. To give two examples out of a rule set of 500 rules currently in use, the assignment of vector objects with different index directions which points to a potential MSB/LSB mismatch and a latch which points to an unwanted piece of hardware resulting from a missing assignment in a conditional branch. The formal assertion checking proves the model inherent assertions as range overflow in VHDL, the design inherent restrictions as bus contention, and the explicitly in-lined assertions. More concrete, the tool tries to find a violation of one of the assertion by checking all possible combinations of input sequences and flipflop states. The formal property checker works similarly but explicitly formulated properties, written in a specific property language, were proven. Also HDL simulation has not kept the same as 10 years ago. A potential simulation framework is shown in Figure 2. Not ex-

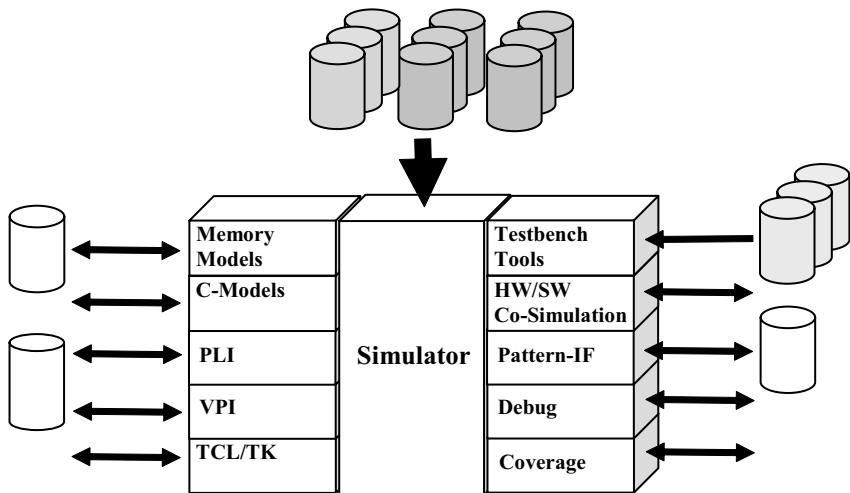


Figure 22.2. Todays Simulation Environment

Implicitly shown, the two HDLs VHDL and Verilog must be supported in parallel. Additionally, memory models, other C-models as instruction set CPU models, and user made C-code may be linked to the simulator. For consideration of embedded software, HW/SW co-simulation tools are needed. Proprietary interfaces as well as the standardized PLI (for Verilog) and VPI (for VHDL) may be used. For advanced tool usage, application specific TCL/TK scripts or specific debugging must be supported. Last but not least, specific testbench languages are essential for modern simulation based verification because HDLs lack of abstraction, miss constraint and unconstraint randomization, and weak support of temporal expressions for assertions and coverage points. Clearly, all tools and models must be interoperable.

Both, the picture of the modern verification flow and the picture of the current simulator usage give a feeling about the diversity of documents and formats currently used for verification. Figure 3 summarizes the languages, sorted by abstraction (System, RTL, Gate) and application (Verification, Design). Altogether seven languages are in use, namely: C, C++, SystemC, E or Vera, VHDL, and Verilog. The variety of languages increases if subsets resulting from different kind of tools (linting, simulation), different tool products (e.g., Modelsim from Mentor, VCS from Synopsys, NC-Sim from CADENCE), different language versions (e.g., VHDL'87, VHDL'93) and also synthesis (RTL subset) are considered. The lack of a clear definition of the interfaces between the languages

	Verification		Design	
System	C/C++ SystemC HVLs		C/C++ SystemC VHDL Verilog SystemVerilog	
RTL	E VERA <i>VHDL</i> <i>Verilog</i> SystemVerilog	PSL/Sugar OVL <i>SystemC</i>	SystemC <i>VHDL'87</i> <i>VHDL'93</i> <i>VHDL'0</i>	'95 Verilog'01 SystemVerilog 3.0 SystemVerilog 3.1 SystemVerilog 3.1b Verilog'07
Gate	<i>VHDL</i> Verilog SystemVerilog		<i>VHDL</i> Verilog SystemVerilog	

Figure 22.3. Languages used in Verification and Design

makes the situation even worse. The story of “Voices from the tower of Babel, Part II” is closed to be finished.

The drawbacks to the current design processes and design flows are manifold. A lot of costs sums up and increases the design cost:

- All designers and verification engineers must learn a set of languages. Cost for trainings come up.
- Several design iterations must be made due to incompatibility of languages or subsets of languages.
- A huge tool-zoo must be bought and prepared. Long discussion rounds with one or more EDA vendors need to be set up to bring clarification in tool interoperability.

It is obvious to say, that the presented situation in verification is hardly acceptable and calls for simplification. Key for simplification is mainly the reduction of languages used in the design flow. Reduction of tools may follow later on, if more than one method is supported by a single tool (e.g., one tool for lint, assertion checking, and property checking). Replacement of all hardware design languages (VHDL, Verilog), the hardware verification languages (E, VERA), and partially also the system languages (C/C++/SystemC) by SystemVerilog is a promising option. In the next section, features of SystemVerilog are described and related to different tasks in a modern design flow.

## 2. Features of SystemVerilog

**Support of new design techniques.** As already mentioned, a set of different design methods is needed to cope with the continuously increasing complexity of the verification task. SystemVerilog's intention is to support most of the existing and new methods (e.g. interface based design, random constraint generation, property checking). For that purpose we summarize in this section a lot of the new features sorted by their major application. First the next two chapters summarize the benefits of SystemVerilog for synthesis. Afterwards, three verification specific topics are presented. Finally, some goodies already available in VHDL, but new in Verilog and interfaces to other languages were discussed.

**Interface based Design.** Interfaces are one of the big advantages of SystemVerilog by improving the VHDL or Verilog practice of structural design. They were taken conceptually from Superlog.

Interfaces bundle is a set of signals, which forms the physical implementation of a protocol. This is similar to a record. Interfaces optionally provide access methods to its objects, which are close to methods in OO-programming. Interfaces also may have parameters, which are close to generics for entities. Interfaces also allow for composition of new interfaces from objects and other interfaces. Additional to that, interfaces allow the specification of so-called modports. Modports make a part of the objects of an interface visible and define access rights (in, out, inout) for each object. Finally, modports can be instantiated in a module interface list. Here, they represent a set of single port maps, one for each single object of the interface. The port direction is taken from the access rights associated with the objects in the modport. Over that, interfaces allow for the specification of assertions. The assertion's temporal logic expression is continuously checked and guarantees a valid state of the objects and a valid access to the objects. The application of interfaces is primarily the definition of hardware for standard protocols.

**Other Enhancements for Synthesis.** Besides interfaces, which enhance description capabilities also for synthesis, SystemVerilog includes a set of very useful constructs. Borrowed from C, increment (++) and decrement (--) operators as well as a wide set of assignment operators (e.g., +=) is available now in SystemVerilog. Packed arrays and records are also new. They allow to combine any number of bits (resulting from the single array or record objects) to one word. Packed unions give SystemVerilog the modeling capability, which VHDL gets from object alias: The multiple naming of a piece of data. To improve clearness of synthesis descriptions, SystemVerilog included unique and priority decision statements

(i.e., if-statement and case-statement). The error prone full-case and parallel-case pragmas are superfluous now. SystemVerilog also extended the general always-block with always\_ff-block, always\_comb-block, and always\_latch-block. According to their naming, they guarantee that the modeled RT-block has glue logic with output flipflops, glue-logic only, or glue-logic mixed up with latches. All SystemVerilog tools must check that behavior to detect of slipped-in errors early.

**Assertion-based Verification.** Assertions are constructs that help to verify the design behavior. They are statements, which analyze an expression or a temporal sequence of an expression. Assertions prove that the assertions hold for true. They have the capability to give a message to the user. Subsequently, a simple example is shown:

```
always @(negedge clk)
if ( ! reset )
assert one_hot( state_vector );
```

Here, the one\_hot encoding of a state\_vector is checked every clock cycle. Assertions can also be used to collect information, for instance how often the Boolean expression held or how often this expression was false. A similar expression, the coverage expression, can also be used to collect such information. However, the collection of the coverage is done automatically and can be interfered with other coverage information. Compared to Verilog'2001 and to VHDL, the SystemVerilog assertions are more powerful, because also output signals can be read. SystemVerilog assertions have also more capabilities, because several constructs are included to build an assertion.

**Formal Verification.** Assertions not only offer to be used for simulation, but also for formal verification. Already, several tools exist to formally check invariants described by the designer using in-lined assertions. Because SystemVerilog, as Verilog, allows for accessing objects over hierarchy boundaries, assertions can also be described outside a module. This makes them usable for verification engineers, because they can write assertions independently from the designer. The support of implications and conditions in assertions give the chance to formulate assertions under specific conditions. This kind of assertions, often called properties, can be used as input for a property checker. Modern Testbench Support The extensions of SystemVerilog towards testbenches are manifold. Plenty of them result from an adoption from Synopsys' VERA donation to the SystemVerilog standardization committees. For specification of the testbench, a so called program block was inserted. This program

block can contain initial blocks only, this means they are activated only once. Specific for this block is the capability to associate one or more clocking domains to it. In this way, clocked delays inside the program block are related to that clock. Furthermore, strobe time points and delay time points relative to the clock can be specified for signals, which are read or written, respectively, inside the block. To support the upcoming constraint-random verification strategy, SystemVerilog has features for random (already in Verilog) and constraint random generation as well as for property and value coverage specification. Testbench modeling is supported by many new constructs and concepts known from software development. They rank from simple things as string data types, enhanced loops, and associative arrays classes over enhanced subprograms and enhanced fork-join statements to classes and inter process communication via semaphores and mailboxes. To support all this features, the simulation core algorithm of SystemVerilog was enhanced and supports now also final blocks, which are executed at the end of simulation.

**Goodies known from VHDL.** SystemVerilog also shows several goodies known from VHDL but up to now not available in Verilog. This includes packages, which help to structure the SystemVerilog code and to give a possibility for collection of declarations. Two VHDL features, but already available for Verilog 2001, shall be mentioned here also: libraries and configurations. Next, a richer set of data types, especially 2-state types ('0','1') are available. They include the bit and in (32-bit) type from VHDL but also the byte (8-bit), shortint (16-bit), and the longint (64-bits) type. Also user-defined types incl. enumeration types and structures are now available in SystemVerilog. Beyond that, unions, dynamic arrays, and associative arrays can be used for modeling.

**Easy Interface to other Languages.** Besides the more tool-oriented PLI, SystemVerilog offers now a DPI, a so-called Direct Programming Interfaces. Here, C-functions can be called directly. Because the type system of SystemVerilog is quite close to the C-type system, type or value adoptions need not be performed at the interface.

### 3. Challenges

One driver of SystemVerilog is to have a single language for design and verification supporting all design levels from gate/switch level to system level. The single language approach helps to reduce the number of design languages. It also defines the interface semantic of different abstract models by having a commonly defined simulation algorithm for

all constructs in the language. Clearly, such a general approach also may have some drawbacks.

- First, the single language solution produced a highly complex hardware design and verification language (HDVL). Over that, the SystemVerilog approach in putting everything in the language increases complexity even more. The approach of using natively implemented code such as packages in VHDL or Java for defining parts of the language was not used in SystemVerilog. Taking the story of the tower of Babel as comparison again, SystemVerilog might solve the confusion of today's multi-language design by coming up with something like the tower of Babel. Following the tale of the story, the language builders will start to talk to each other in another tongue. Surprisingly, this can be partially observed already today: For the next IEEE Verilog standard competing solutions were discussed, which might lead to a completely different language concept for Verilog 2005 than for SystemVerilog. Also, upcoming tools very probably will support different SystemVerilog subsets for Synthesis and formal Verification.
- Second, the innovation speed was achieved only by putting or adopting, under high time pressure, concepts from other languages as Superlog, C++, VHDL, or VERA in SystemVerilog. The result was lack of quality, which could be seen in the 3.1. and 3.1a updates and corrections of the standard. The result was also a heterogeneous language, with less orthogonal concepts.
- Third, no pilot tool was developed in conjunction with SystemVerilog. So, there was experience in using parts of the concepts. Using the complete language lacks of experience. Issues, which are currently missing are modports from interfaces, which can be passed and refined for lower hierarchies or implication, with full independence of the assume part and the prove part.

These issues might but must not become critical for SystemVerilog's introduction.

- The subset issue can be solved by standardization of the subsets, as already done in VHDL's synthesis subset. Also a clear interface to VHDL must be defined to keep connection to existing VHDL legacy code.
- The innovation speed is also a benefit for the language. Instead of five years (or more) theoretical discussions on one standard, as usual in standards bodies, every year, a new standard document

was finished. EDA vendors started implementation on each standard document and feed implementation issues directly in the next standard. EDA tools supporting at least a subset of SystemVerilog are available to users.

- Their gathered experience can be feed back to the next standard, which then might be already an IEEE standard.

## 4. Summary

We presented design-flow-related motivation for SystemVerilog. After that, we presented some technical issues of SystemVerilog, which show that also other improvements can be expected from SystemVerilog “in-use”. We also showed some challenges of SystemVerilog, which should be overcome by continuation of SystemVerilog enhancement and by subset standardization. Unfortunately, mainly US-EDA vendors contributed to the standardization. We guess that contributions from user’s side and from academia are necessary. Also, European and Asian-Pacific researchers and developers shall contribute to the work.

## Bibliography and Resources

The best source for SystemVerilog related is of course the SystemVerilog home page [www.systemverilog.org/](http://www.systemverilog.org/). Especially technical papers including a continuous update, trainings material and tutorials are also available in the SystemVerilog home page under <http://www.systemverilog.org/techpapers/techpapers.html>. A free copy of the SystemVerilog LRM is also available ([http://www.eda.org/sv/SystemVerilog\\_3.1\\_final.pdf](http://www.eda.org/sv/SystemVerilog_3.1_final.pdf)). An online seminar can be ordered at [www.systemverilognow.com/](http://www.systemverilognow.com/). SystemVerilog seminars can also be booked by major EDA related trainings firms. SystemVerilog-related information is made available also on the www-pages of the major EDA companies. Actual information can also be gathered on the www-page of ee-times ([www.eet.com](http://www.eet.com)). The citation “Voices from the Tower of Babel” was originally a presentation in the pre-VHDL days (G. Jack Lipovski: “Hardware description languages: Voices from the Tower of Babel”, Computer, Volume 10, Number 6, June, 1977, pp14-17). The authors summarized the numerous HDLs in use, described the incompatibility of the EDA systems and models and motivated the task force towards a single standardized language.